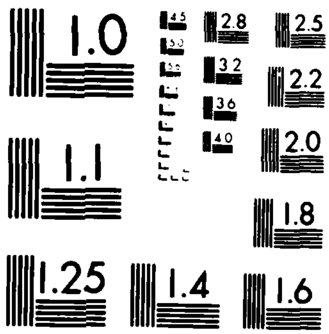
AD-A150 584 PROCEEDINGS OF THE TECHNICAL FORUM (3RD) ON THE F-16 MIL-STD-1750A MICROP..(U) AERONAUTICAL SYSTEMS DIV WRIGHT-PATTERSON AFB OH J L PESLER ET AL. 06 MAY 82 1/6

UNCLASSIFIED ASD-TR-82-5011-VOL-2 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

ASD-TR-82-5011

(2)

AD-A150 584

THE THIRD TECHNICAL FORUM

ON THE

F-16 MIL-STD-1750A MICROPROCESSOR

AND THE

F-16 MIL-STD-1589B COMPILER

5 - 6 MAY 1982

AIR FORCE MUSEUM AUDITORIUM
AREA B
WRIGHT-PATTERSON AFB OHIO 45433

PROCEEDINGS

VOL II OF II: SPECIFICATIONS

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

DTIC FILE COPY

DTIC
S ELECTE D
FEB 1 5 1985
E

85 01 30 044

NOTICE

When Government drawings, specifications or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Public Affairs office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Jeffery L. Pesler
JEFFERY L. PESLER
Project Engineer
Deputy for F-16

Felix O. Glover
FELIX O. GLOVER
Dep Chief, Avionics Division
Deputy for F-16

J. P. Brailey
JOHN P. BRAILEY
Director of Engineering
Deputy for F-16

George L. Monahan
GEORGE L. MONAHAN, JR.
Brigadier General, USAF
System Program Director
Deputy for F-16

"If your address has changed or if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify ASD/YPEA, W-PAFB, OH 45433 to help maintain a current mailing list".

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

ASD-TR-82-5011
Vol II of II

PROCEEDINGS: SPECIFICATIONS
OF THE
THIRD TECHNICAL FORUM
ON THE
F-16 MIL-STD-1750A MICROPROCESSOR
AND THE
F-16 MIL-STD-1589B COMPILER

5-6 May 1982

AIRFORCE MUSEUM AUDITORIUM
AREA B
WRIGHT-PATTERSON AFB OHIO 45433

Approved for Public Release;
distribution unlimited.

Sponsored by: F-16 SYSTEM PROGRAM OFFICE

Hosted by: F-16 SYSTEM PROGRAM OFFICE

Organized by: AVIONICS ENGINEERING DIVISION (ASD/YPEA)
DIRECTORATE OF ENGINEERING
DEPUTY FOR F-16
AERONAUTICAL SYSTEM DIVISION
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AFB OHIO 45433

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ASD-TR-82-5011	2. GOVT ACCESSION NO. AD-A150584	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Proceedings; Specifications		5. TYPE OF REPORT & PERIOD COVERED Vol II of II 5-6 May 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Editors: Mr Jeffery L. Pesler		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Hq ASD/YPEA Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Hq ASD/YP Wright-Patterson AFB, Ohio 45433		12. REPORT DATE 5-6 May 1982
		13. NUMBER OF PAGES 520
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same as above.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

MIL-STD-1750A; MIL-STD-1589B; Microprocessor; Computer Instruction Set Architecture; Compilers; Support Software; Mnemonics; Digital Avionics; Standardization.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a collection of unclassified specifications to be distributed initially at the Third Technical Forum on the F-16 MIL-STD-1750A microprocessor and support software. The purpose of this forum is to keep interested organizations informed of the progress of the F-16 embedded computer standardization efforts.

FOREWORD

The Deputy for F-16 is sponsoring the development of a MIL-STD-1750A microprocessor and MIL-STD-1589B support software in conjunction with the Multinational Staged Improvement Program (MSIP). For the next two days we will provide a comprehensive technical summary of the F-16 embedded computer standardization efforts and a forum of papers from other "users". Significant interest in the F-16 MIL-STD-1750A/MIL-STD-1589B programs have prompted this technical forum.

The purpose of the Third Technical Forum is to present the technical details of the F-16 embedded computer standardization efforts, to exchange ideas on lessons learned from other system applications and present other hardware/software developments in support of these standards.

This is the SPECIFICATION Volume, Volume II of the Third Technical Forum Proceedings. Volume I will contain the papers presented during the conference. It will be mailed to the attendees at a later date.

Many thanks to Mr. Jeffery L. Pesler for his key role and leadership in the adoption of the standards within the F-16 MSIP as well as organizing and directing the Third Technical Forum. Also to the ASD/ENA and ASD/AX staff who have assisted in organizing the technical program and proceedings, Mr. Jerry L. Duchene and Mr. Ronald S. Vokits. Special thanks to Mr. Randal K. Moore of General Dynamics for his leadership and support.

Thanks also to the moderators and all the speakers, who responded with outstanding presentations and papers in a timely manner, despite such short notice. And, finally, to the secretaries Mrs. Marie Jankovich, Mrs. Sharlene Thompson, Mrs. Brenda Harris and Mrs. Connie Castin for their expert administrative help in handling the conference correspondence and registration.

GEORGE L. MONAHAN, JR.
Brigadier General, USAF
System Program Director
Deputy for F-16

Accession For	
NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

DTIC COPY INSPECTED

PROCEEDINGS

THIRD TECHNICAL FORUM

ON THE

F-16 MIL-STD-1750A MICROPROCESSOR

AND THE

F-16 MIL-STD-1589B COMPILER

5-6 May 1982

WRIGHT-PATTERSON AFB, OHIO

PRELIMINARY

SPECIFICATION NO: 16ZE181
CODE IDENT : 81755
DATE : 22 September 1981

CRITICAL ITEM DEVELOPMENT SPECIFICATION

FOR

MIL-STD-1750A MICROPROCESSORS

Contract F33657-75-C-0310

CCP 9105

APPROVED	APPROVED	APPROVED
10/8/81	10-8-81	
P.L.CURRIER GROUP ENGINEER CONTROLS AND DISPLAYS	RON THOMPSON SPECIALITY ENGINEERING	L. CRITTENDEN MANAGER

This document contains Technical Data considered to be a resource under ASPR 1-329.1(b) and DoD Directive 5400.7 and is not a "record" required to be released under the Freedom of Information Act.

LIST OF FIGURES

LIST OF TABLES

1. SCOPE

1.1 This specification establishes the performance, design, test manufacture and acceptance requirements for small, low-power, high performance, and cost effective MIL-STD-1750 Microprocessors.

1.2 Classification. MIL-STD-1750 microprocessors covered by this specification shall be classified by performance and power requirements as follows:

Class I. A Class I designation will apply to a medium performance, low power microprocessor in conformance with paragraph 3.2.1 herein.

Class II. A Class II designation will apply to a high performance higher power microprocessor in conformance with paragraph 3.2.1. herein.

2 APPLICABLE DOCUMENTS

2.1 Government documents. The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered as superseding requirements. However, in the event of a conflict between MIL-STD-1750 and this document, the contents of MIL-STD-1750 shall be considered to be a superseding requirement.

SPECIFICATIONS

Military

MIL-E-5400P 2 July 1973	Electronic Equipment, Airborne, General Specifications for
MIL-T-18303B 1 Sept 1966	Test Procedures; Preproduction and Acceptance for Aircraft Electronic Equipment, Format of
MIL-M-38510D 16 May 1979	Microcircuit Design, General Specification for

STANDARDS

Military

MIL-STD-130D Notice 3 1 Aug 1973	Identification Marking of U.S. Military Equipment
MIL-STD-454D 31 Aug 1973	Standard General Requirements for Electronic Equipment
MIL-STD-461A Change 3 1 May 1970	Electromagnetic Interference Characteristics
MIL-STD-462 Change 2 1 May 1970	Electromagnetic Interference Characteristics, Measurement of
MIL-STD-781B Change 1 28 July 1969	Reliability Tests: Exponential Distribution
MIL-STD-810B 15 Jun 1967	Environmental Test Methods
MIL-STD-883B Notice 4 4 Nov 1980	Test Methods and Procedures for Microelectronics
MIL-STD-891B 1 Apr 1974	Contractor Parts Control and Standardization Program
MIL-STD-1472A 15 May 1970	Human Engineering Design Criteria for Military Systems, Equipment and Facilities
DOD-STD-1686 2 May 1980	Electrostatic Discharge Control Program for Protection of Electrical and Electronic Parts, Assemblies and Equipment (Excluding Electrically Initiated Explosive Devices) (Metric)

MIL-STD-1750A Notice 1, Jul 1981	Sixteen bit instruction Set architecture

OTHER PUBLICATIONS

OSHA Standards

OSHA STD 1910.93 18 Oct 1972	Code of Federal Regulations Part 1910 Occupational Safety and Health Standards

2.2 Non-Government documents. The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of a conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered as superseding requirements. Documents referenced withinthe documents cited herein shall not be applicable to this specification because of such reference.

SPECIFICATIONS

General Dynamics

16PP027 (Current Revision)	Program Parts Selection List

16PP224 14 Jul 1976	Subcontractor/Vendor Packaging and Transporation Instructions for Shipment to General Dynamics/ Fort Worth
16PP379A 7 Jul 1981	Avionic Processor Standard Instruction Set Architecture Requirements

2.3 Other publications. The following publications along with the government source from which they may be obtained are listed below:

No number 30 Apr 1981	Acceptance Test Program for MIL-STD-1750A Sixteen Bit Instruction Set Architecture (available from ASD/ENA SD)
No Number Undated	Support Software for MIL-STD-1750A Sixteen Bit Instruction Set Architecture (available from ASD/XR)

3. REQUIREMENTS

The requirements for MIL-STD-1750 microprocessors, hereinafter referred to as microprocessors, shall be as specified herein.

3.1 Item definition. The microprocessor hardware will be partitioned into three (3) principal components. The first component is the Central Processing Unit (CPU) which will be used for the interpretation and execution of the MIL-STD-1750 Instruction Set Architecture. The second component is an optional Memory Management Unit (MMU) for mapping of logical memory blocks and for extending the addressable memory space and access control. The third component is the Block Protect RAM for protecting system memory in 1K blocks. The Block Protect RAM is optional with either a CPU or CPU and MMU.

3.1.1 Functional diagram. The functional diagram for a CPU with optional Block Protect RAM is shown in Figure 1. The functional diagram for a CPU with optional MMU and Block Protect RAM is shown in Figure 2.

3.1.2 External interface. The microprocessor hardware shall conform to but not be limited by the external signal interface defined herein.

3.1.2.1 Central Processing Unit (CPU). The CPU shall implement the instruction set of the MIL-STD-1750 ISA. The external interface for for the CPU shall include, but not be limited by, the signals defined in Figure 3 which are described below.

3.1.2.1.1 Reset. The microprocessor shall utilize this signal to perform initialization in accordance with paragraph 3.2.1.2.1.1

3.1.2.1.2 CPU clock. The microprocessor shall utilize this signal to determine the time period of all basic functions.

3.1.2.1.3 Timer clock. The microprocessor shall utilize this signal to determine the time period of TIMER A and TIMER B.

3.1.2.1.4 Trigger go reset. As a design goal, the trigger go reset discrete shall be provided to reset the trigger go counter. The trigger go counter is a user option described in paragraph 6.

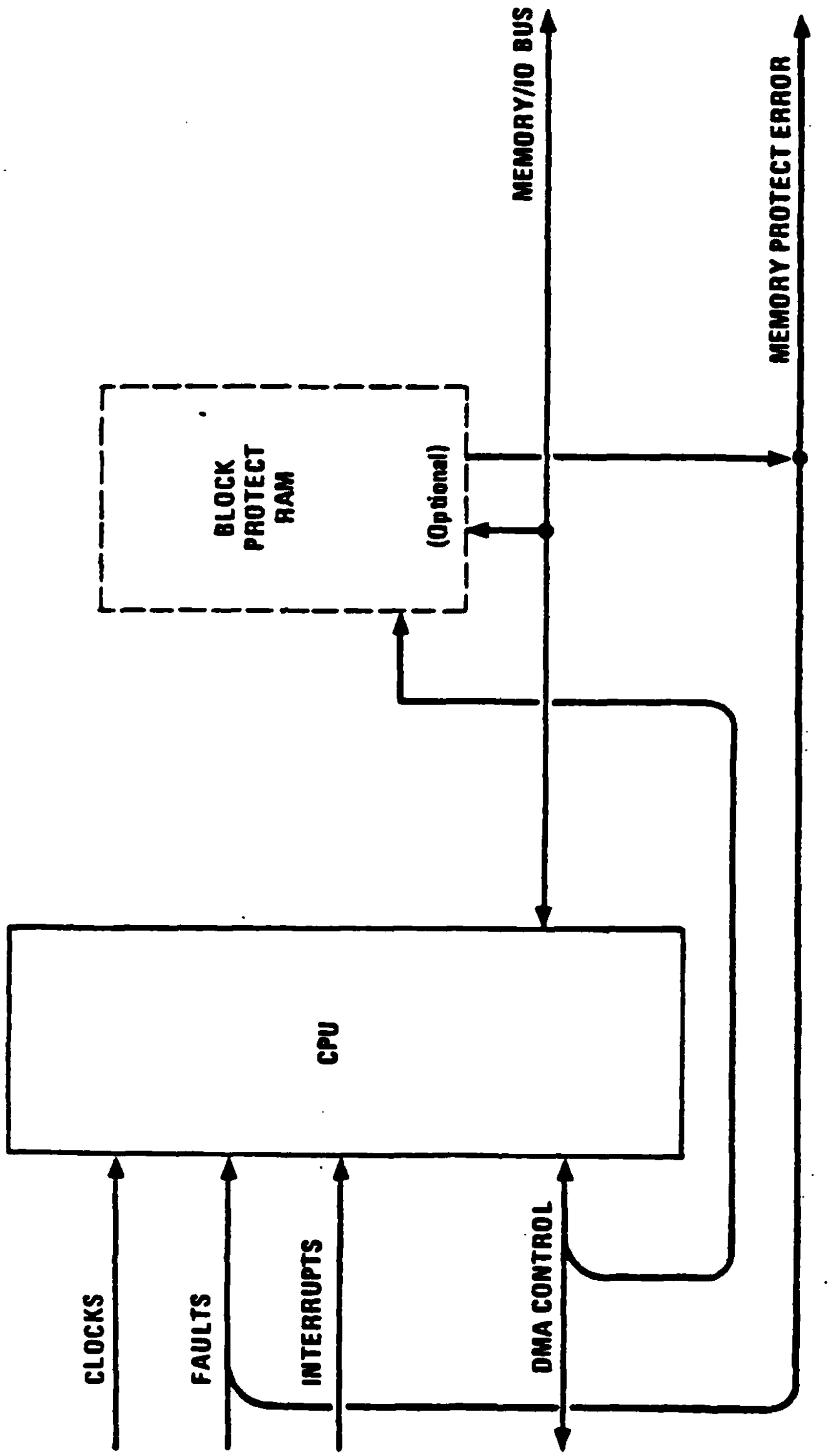


Figure 1 CPU with Optional Block Protect RAM

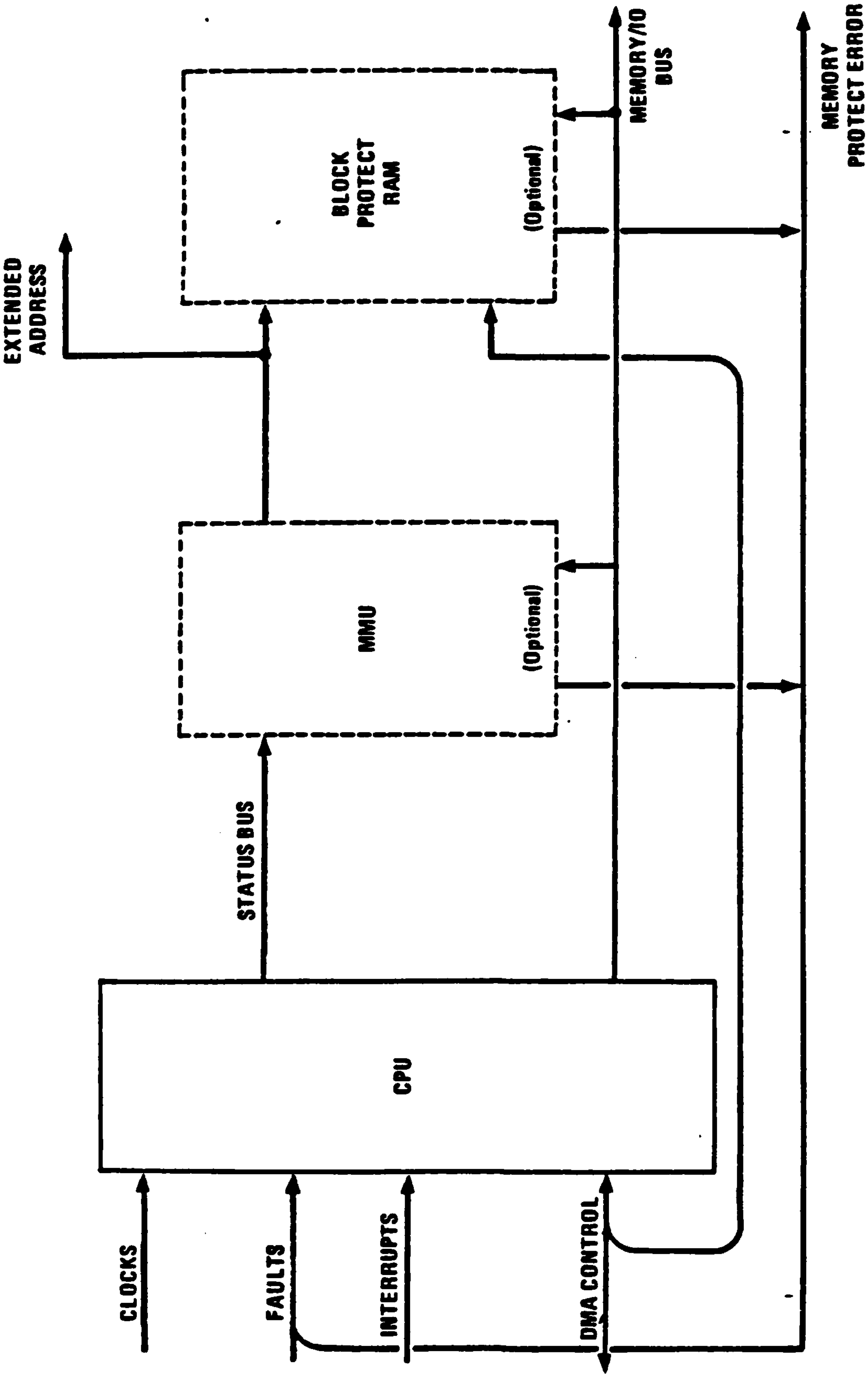


Figure 2 CPU with Optional MMU and Block Protect RAM

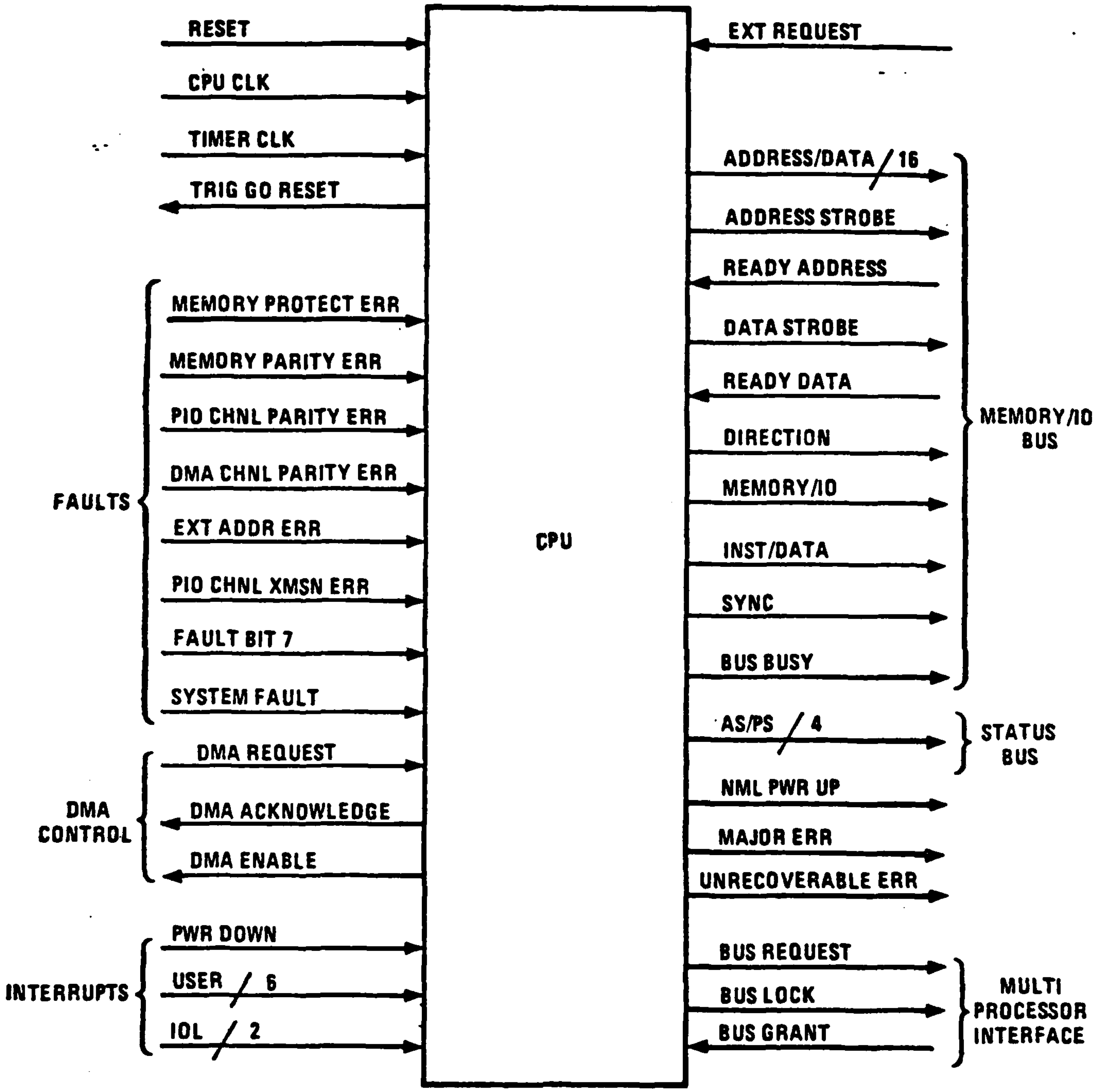


Figure 3 Central Processing Unit

3.1.2.1.5 Faults. The CPU shall be capable of receiving and responding to the following faults from an external source:

(a) Memory Protect Error,
(b) Memory Parity Error,
(c) PIO Channel Parity Error.
(d) DMA Channel Parity Error,
(e) External Address Error,
(f) PIO Channel Transmission Error,
(g) Fault Bit Seven (7),
(h) System Fault,

3.1.2.1.6 DMA control. The CPU shall perform the bus acquisition logic function for direct memory access devices.

3.1.2.1.6.1 DMA request. The CPU shall accept asynchronous DMA bus requests from DMA devices via the DMA request signal.

3.1.2.1.6.2 DMA acknowledge. If enabled by the software, the CPU shall acknowledge DMA requests after it has disabled its own address/data bus.

3.1.2.1.6.3 DMA enable, The DMA enable discrete output shall be provided to indicate the readiness of the CPU to respond to DMA requests.

3.1.2.1.7 Interrupts. The CPU shall accept the following interrupts:

(a) an externally generated Power Down Interrupt
(b) six (6) general purpose interrupts
(c) two (2) input/output level interrupts

3.1.2.1.8 External request. The external request input discrete shall provide as a minimum the following functions in the CPU:

(a) Halt
(b) Disable
(c) Breakpoint
(d) Load and examine registers
(e) Resume program execution

3.1.2.1.9 Memory/IO bus. The memory/IO bus shall contain the necessary signals to interface to external memory and I/O.

3.1.2.1.9.1 Address/data bus (AD0-AD15). The CPU shall utilize a 16-bit time multiplexed address/data bus to identify the location of data in memory or to identify the source of data for an input/output command.

3.1.2.1.9.2 Address strobe. Address strobe will be used to latch memory and XIO addresses.

3.1.2.1.9.3 Ready address. The ready address signal shall be driven by a memory module or an I/O device to signify that data will be valid within the time specified for the CPU class. In the absence of ready address, the CPU shall extend the address cycle to accommodate slower devices.

3.1.2.1.9.4 Data strobe. The data strobe shall be used to gate data from an external device into the CPU or to control the timing of data being written.

3.1.2.1.9.5 Ready data. The ready data signal will be driven high by a memory module or an I/O device to signify that data will be valid within the time specified for the CPU. In the absence of ready data, the CPU shall extend the data transfer cycle to accommodate slower access time devices.

3.1.2.1.9.6 Direction. The direction signal shall specify data transfer cycles as read or write with respect to the CPU. Split cycles or read modify write cycles shall not be used.

3.1.2.1.9.7 Memory/IO. This control line shall differentiate between memory and input/output references.

3.1.2.1.9.8 Instruction/data. This control line shall differentiate between instruction and data references to memory.

3.1.2.1.9.9 Sync. This signal may be used for external synchronization of memory and I/O control.

3.1.2.1.9.10 Bus busy. This signal shall establish the end of a bus cycle. The trailing edge of this signal is used to sample bits into the fault register.

3.1.2.1.10 Status bus (AS/PS). The CPU shall utilize a 4-bit status bus to transfer address state and processor state information from the CPU status word to the MMU.

3.1.2.1.11 Normal power up. The normal power up discrete shall be set when power to the CPU has completed the transition from less than operational to operational level.

3.1.2.1.12 Major error. The major error discrete shall be set when the CPU machine error interrupt is masked and a fault is reported indicating that the program is executing out of its expected software environment, (Reference Table VI, Fault Detection Criteria, External).

3.1.2.1.13 Unrecoverable error. The unrecoverable error discrete shall be set when the CPU machine error interrupt is masked and a fault is reported indicating that the next instruction cannot be reliably found and/or executed, (Reference Table VI, Fault Detection Criteria, External).

3.1.2.1.14 Multiprocessor interface. The following signals are implemented for bus arbitration when connecting CPU's in a tightly coupled multiprocessor configuration.

3.1.2.1.14.1 Bus request. The bus request shall indicate that a new bus cycle is required.

3.1.2.1.14.2 Bus lock. The bus lock shall be used by the requesting device to obtain successive cycles.

3.1.2.1.14.3 Bus grant. The bus grant shall indicate the bus is available for use by another device.

3.1.2.2 Memory management unit (MMU). The MMU shall implement the appropriate instructions of the MIL-STD-1750 ISA. In addition, the MMU shall convert the instruction/data status, address state (AS), and the four (4) most significant bits of the logical address into the 8-bit Extended address. When the resulting extended address is appended to the remaining 12 bits of logical address the result is a 20-bit physical address. The external interface for the MMU shall include, but not be limited by, the signals shown in Figure 4 which are described below.

3.1.2.2.1 Memory/IO bus. The memory/IO bus for the MMU shall connect directly to and be compatible with the memory/IO bus for the CPU as described in paragraphs beginning at 3.1.2.1.

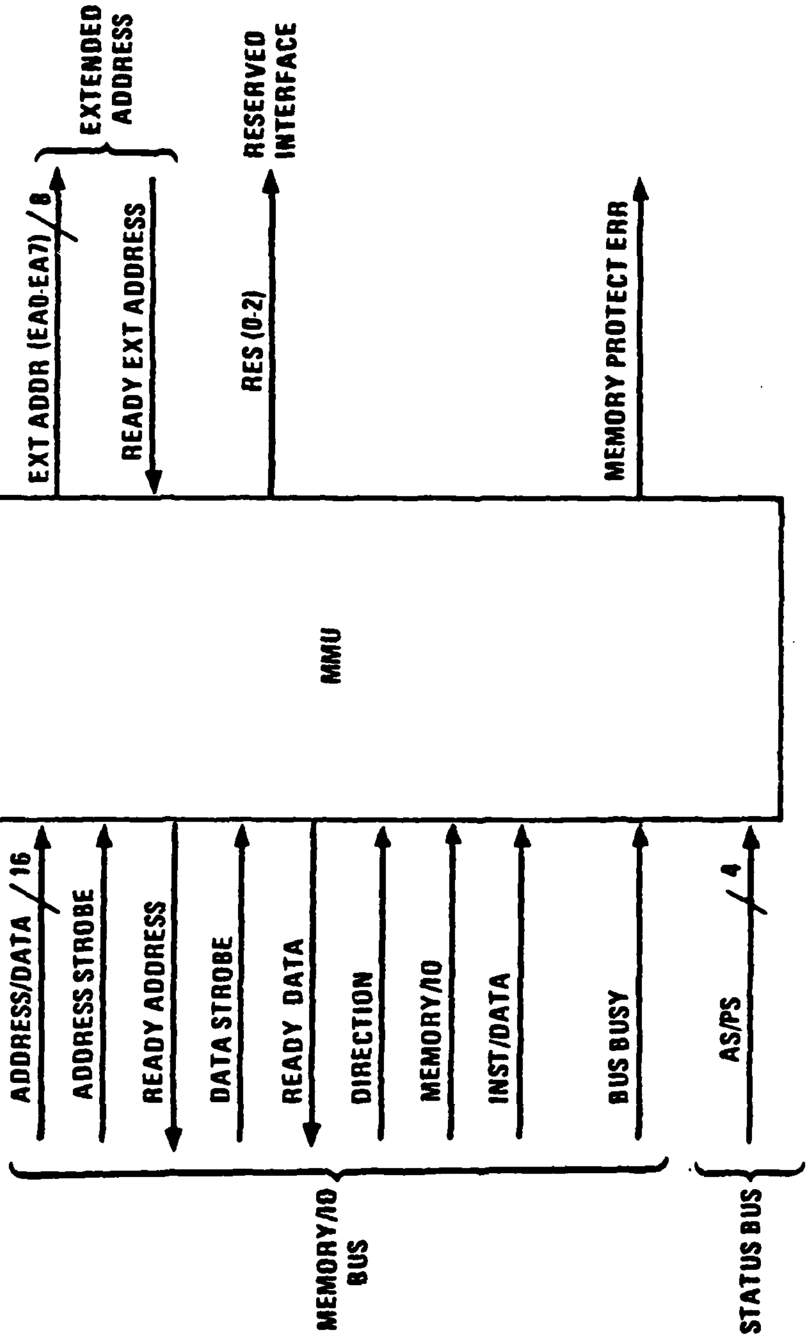


Figure 4 Memory Management Unit

12

3.1.2.2.2 Status bus (AS/PS). The status bus for the MMU shall connect directly to the status bus for the CPU as described in paragraphs beginning at 3.1.2.1.10.

3.1.2.2.3 Extended address. The extended address shall provide the eight (8) most significant bits of physical address and a ready extended address to enable extending the time the address is available for a memory or an I/O device.

3.1.2.2.3.1 Extended address bus (EA0-EA7). The extended memory address bus shall increase the memory addressing range to 1,048,576 words. The extended address bus shall be defined by the 8 physical page address bits from the MMU page register defined in paragraph 3.1.3.2.

3.1.2.2.3.2 Ready extended address. The ready extended address signal shall replace the ready address signal defined in paragraph 3.1.2.1.9.3 for the purpose of extending the address time on the bus.

3.1.2.2.4 Reserved interface (R0-R2). The reserved interface shall be defined by the reserved bits in the MMU page register definition in paragraph 3.1.3.2.

3.1.2.2.5 Memory protect error. The memory protect error signal shall be used by the MMU to report access errors, execute protect errors, and operand write protect errors to the CPU through the memory protect fault discrete defined in paragraph 3.1.2.1.5.

3.1.2.3 Block protect RAM (BPR). The block protect RAM shall provide the option of protecting portions of the system memory from access both by CPU and DMA. The external interface for the block protect RAM shall include, but not be limited by, the signals defined in Figure 5 which are described below.

3.1.2.3.1 Block protect RAM extended address bus (BA0-BA7). The block protect RAM extended address bus shall interface to the logical or extended address bus as defined in Table I.

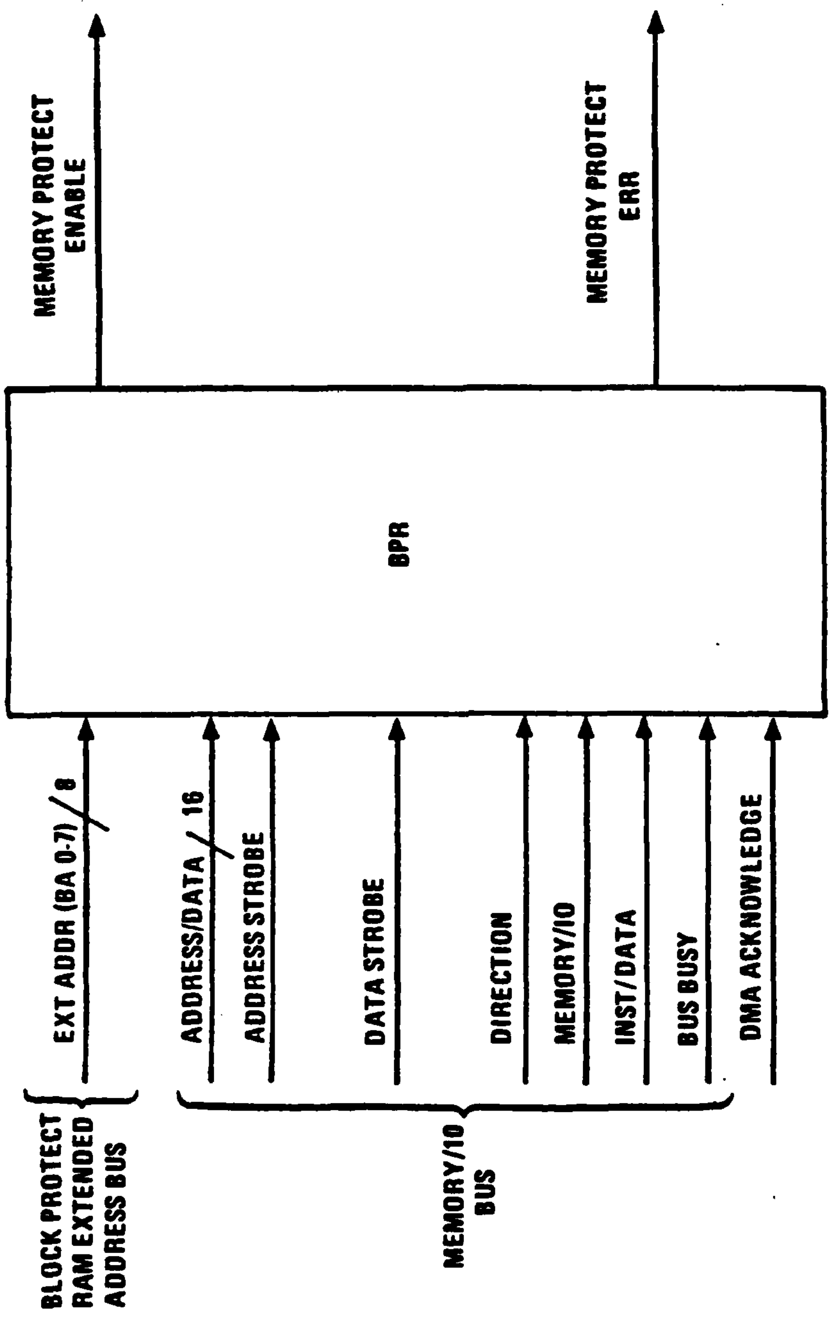


Figure 5 Block Protect RAM

TABLE I
BLOCK PROTECT RAM ADDRESS BUS INTERFACE DEFINITION

BPR ADDRESS	BA0 - BA3	BA4 - BA7
CPU	0 - 0	AD0 - AD3
EXTENDED MMU ADDRESS	EA0 - EA3	EA4 - EA7

3.1.2.3.2 Memory/IO bus. The memory/IO bus of the Block Protect RAM connects directly to the memory/IO bus of the CPU described in paragraphs beginning at paragraph 3.1.2.1.9.

3.1.2.3.3 DMA acknowledge. The DMA acknowledge shall connect to the CPU DMA acknowledge defined in paragraph 3.1.2.1.6.2.

3.1.2.3.4 Memory protect enable. This discrete output shall indicate the status of memory protection in the Block Protect RAM.

3.1.2.3.5 Memory protect error. This signal shall be used by the Block Protect RAM to report write protect violations of the CPU or the DMA to the CPU through the Memory Protect Fault discrete described in paragraph 3.1.2.1.5.

3.1.3 Internal interface. The internal interface shall be implemented as described herein.

3.1.3.1 CPU internal interface. The CPU internal interface shall be implemented as described herein. The CPU shall make use of the following registers: instruction counter, general registers, status word, fault register, interrupt mask, pending interrupt register, interval timer, and trigger go counter.

3.1.3.1.1 Instruction counter (IC). A 16-bit register shall be used for program sequencing within a range of 65,536 words. It shall be external to the general registers. It shall be saved in memory when an interrupt is serviced.

3.1.3.1.2 General registers (RA). Sixteen bit general purpose registers shall be provided for use as accumulators, index registers, base registers, temporary operand memory, and stack pointers. For instructions requiring a 32-bit operation, adjacent registers shall be concatenated to form effective 32-bit registers. Instructions requiring a 48-bit operation shall concatenate three adjacent registers to form an effective 48-bit register.

When registers are concatenated, the register specified by the instruction shall represent the most significant word. The register set wraps around, i.e., register 15 concatenates with register 0 for 32 bit operations, and with registers 0 and 1 for 48-bit operations.

3.1.3.1.3 Processor status word (SW). The CPU shall reference a 16-bit processor status word register whose state is defined by some prior event occurrence in the computer. The figure below indicates the format for the SW with the following paragraphs describing the meaning of the Condition Status (CS) field, reserved bits, the Processor State (PS) field, and the Address State (AS) field.

CS				RESERVED				PS				AS			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

3.1.3.1.3.1 Condition status (CS). A four bit field (bits 0 through 3) of the status word shall be dedicated to instruction results (i.e., instruction status bits) and is defined as condition status (CS). Bits 0, 1, 2, and 3 shall be identified as C, P, Z, and N, respectively, and their meanings are given by the following register transfer description:

C = (CS)0 = 1 if result generates a carry from an addition or no borrow from a subtraction.

P = (CS)1 = 1 if result is greater than zero

Z = (CS)2 = 1 if result is equal to zero

N = (CS)3 = 1 if result is less than zero

3.1.3.1.3.2 Reserved bits. Bits 4 through 7 of the status word shall always be zero.

3.1.3.1.3.3 Processor State (PS). A four bit field (bits 8 through 11) of the status word shall be dedicated to the processor state code. The processor state code shall determine the legal/illegal criteria for privileged instructions. When interfaced to an MMU, the processor state code will be used as a memory access key for all instructions and operand references to memory.

3.1.3.1.3.4 Address state (AS). A four bit field (bits 12 through 15) of the status word shall be dedicated to the address state code. When interfaced to an MMU, the address state code shall select the page register group to be used for mapping all instruction and operand references to memory. Otherwise the address state code shall remain zero.

3.1.3.1.4 Fault register (FT). The fault register is a 16-bit register used for indicating machine error conditions. The logical OR of the fault register bits is used to generate the machine error interrupt. The fault register shall be read and cleared by an XIO instruction. If a particular fault bit is not implemented, then the bit shall be set to zero. The fault bits shall be assigned as specified in the following:

MEMORY PROTECT		PARITY			I/O		TG	FAULTS				RES	BITE		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The bits shall have the following meaning when set to one (1):

Bit 0: CPU Memory Protect Fault. The CPU has encountered an access fault, write protect fault, or execute protect fault.

Bit 1: DMA Memory Protect Fault. A DMA device has encountered as access fault or a write protect fault.

Bit 2: Memory Parity Fault.

Bit 3: PIO Channel Parity Fault.

Bit 4: DMA Channel Parity Fault

Bit 5: Illegal I/O Command Fault. An attempt has been made to execute an XIO command which is not implemented by the microprocessor and for which the user subsystem has asserted an External Address Error.

Bit 6: PIO Transmission Fault. Other I/O error checking devices, if used, will be OR'ed into this bit to indicate an error.

Bit 7: Spare. This bit shall be set by the CPU in response to activation of the FT Bit 7 discrete signal from the host subsystem.

Bit 8: Illegal Address Fault. This bit shall be set to denote that the External Address Error discrete has been activated on a memory data access cycle.

Bit 9: Illegal Instruction Fault. An attempt has been made to execute a reserved code.

Bit 10: Privileged Instruction Fault. An attempt has been made to execute a privileged instruction with PS $\neq 0$.

Bit 11: Address State Fault. An attempt has been made to establish an AS value greater than zero in a CPU only configuration.

Bit 12: Bit 12 shall always be zero.

Bit 13: Built-in Test Fault. An error has been detected in the CPU built-in test equipment, or a System Fault has been detected in the host equipment.

Bit 14: Bit 14 shall always be zero.

Bit 15: System Faults. This bit shall be set in response to activation of the System Fault discrete. This bit shall also cause Bit 13 to be set.

3.1.3.1.5 Pending interrupt register (PI). The sixteen bit pending interrupt register shall contain a bit for each system interrupt as defined in Table II. A bit in the pending interrupt register is set independently of the interrupt mask register for each active system interrupt request.

3.1.3.1.6 Interrupt mask (IM). The sixteen (16) bit interrupt mask register shall provide the capability to selectively disable system interrupts as defined in Table II.

3.1.3.1.7 Interval timers. Two sixteen (16) bit interval timers shall be provided in the CPU and shall be referred to as Timer A and Timer B. The clock period for Timer A shall be set by the Timer Clock input to the CPU. The clock period for Timer B shall be derived from the same source and shall be ten (10) times the clock period for Timer A.

3.1.3.1.8 Interrupt definitions. The priority definitions of the interrupts and their required relationship to the interrupt mask and interrupt pointer addresses are illustrated in Table II, Interrupt Definitions.

3.1.3.1.8.1 Machine error. The machine error interrupt cannot be disabled but can be masked during normal operation of the CPU.

3.1.3.1.8.2 Arithmetic exceptions. If a floating point overflow/underflow or fixed point overflow condition occurs, then the instruction generating that condition shall be interrupted at its completion if the interrupt is unmasked and enabled.

3.1.3.1.8.3 Executive call. The executive call interrupt, used with the Branch to Executive instruction, BEX, shall not be masked or disabled.

3.1.3.1.8.4 Power down. The power down interrupt shall initiate the power down sequence and cannot be masked or disabled during normal operation of the computer.

3.1.3.1.8.5 General purpose interrupts (User 0-5). The microprocessor shall implement six (6) general purpose interrupts which shall be available to the using subsystem.

3.1.3.1.8.6 Input/output level interrupts. Input/output level 1 and level 2 interrupts shall be available to the user. Either interrupt level or both may be implemented for an interface as defined by the particular application specification.

TABLE II INTERRUPT DEFINITIONS

Interrupt Number	MK/PI Register Bit Number	Interrupt Linkage Pointer Address (Hex)	Interrupt Service Address (Hex)	
0	0	20	21	Power Down (cannot be masked or disabled)
1	1	22	23	Mac.. ne Error cannot be disabled)
2	2	24	25	User 0
3	3	26	27	Floating Point Overflow
4	4	28	29	Fixed Point Overflow
5	5	2A	2B	Executive Call (cannot be masked or disabled)
6	6	2C	2D	Floating Point Underflow
7	7	2E	2F	Timer A
8	8	30	31	User 1
9	9	32	33	Timer B
10	10	34	35	User 2
11	11	36	37	User 3
12	12	38	39	Input/Output Level 1
13	13	3A	3B	User 4
14	14	3C	3D	Input/Output Level 2
15	15	3E	3F	User 5

NOTE: Interrupt number 0 has the highest priority. Priority decreases with increasing interrupt number.

These interrupts will be used in conjunction with the Input/Output Interrupt Code registers to provide an I/O channel address to process communications. Two levels of interrupt allow easy differentiation of normal reporting from error reporting.

3.1.3.2 Memory management unit registers. The MMU shall provide 512 registers to specify the acceptable/unacceptable conditions for memory references and to perform a memory mapping function from logical memory addresses to physical memory addresses. The memory mapping function shall specify a correspondence between 4096 word blocks if logical memory with 4096 word blocks of physical memory. The following figure illustrates the page register format.

AL				E/W RESERVED					PPA						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

AL Field:

A 4-bit field (bits 0 through 3) of each page register shall contain the access lock (AL) code for the associated page register, which shall be used with the access keycodes to determine access permission.

E/W Bit:

Bit 4 shall determine the execute protect and write protect status for instruction and data references respectively.

Reserved Bits:

Bits 5 through 7 of all of the page registers shall be reserved and shall always be 0.

PPA Field:

An eight-bit field (bits 8 through 15) of each page register shall be dedicated to the physical page address of memory which is associated with the page address register.

3.1.3.3 Block Protect RAM Registers. The Block Protest Ram shall contain up to 128 16-bit Block Protect RAM Registers. Each bit in these registers provides write protection for a 1024-word memory block in the Block Protect RAM. A "0" bit permits writing and a "1" bit provides write protection.

Bit 0 of each Block Protect RAM Register represents the lowest 1024-word block and bit 15 the highest. A single 16-bit register will thus contain bits to provide protection for a total of 16,384 words. The Block Protect RAM Registers 0 thru 63 control write protection for the CPU, and Block Protect RAM Registers 64 thru 127 control write protection for DMA.

3.2 Characteristics.

3.2.1 Performance. The performance characteristics of the microprocessor Set Architecture (ISA) shall be as defined in MIL-STD-1750. The selection of options outlined in MIL-STD-1750 shall be in accordance with 16PP379 and as detailed below. It shall be the responsibility of the supplier to demonstrate conformance with the MIL-STD-1750 ISA to the Air Force and to General Dynamics. Unless otherwise specified, these performance specifications shall apply under the conditions of MIL-E-5400.

3.2.1.1 Throughput. CPU throughput shall be measured using the instruction set mix specified in Table IV. The throughput unit of measure is "thousands of instructions per second" or "KIPS." The CPU throughput shall be in accordance with Table III when interfaced with a memory whose characteristics are as listed.

TABLE IV

PERCENTAGE INSTRUCTION SET MIX - THROUGHPUT

	Single Precision	Double Precision	Floating Point	Extended Floating Point	Total
LOAD/STORE					
Reg	4.7	2.0			
Direct	12.0	2.5		0.2	
Indirect	3.0	0.5			
Immed.	2.4				
Immed. Short	5.9				
Base Rel	11.2	3.0			47.4%
ADD/SUBTRACT					
Reg	2.5	0.4	1.4	0.05	
Direct	0.6	0.6	1.5	0.05	
Immed.	0.2				
Immed. Short	0.6				
Base Rel	2.5		2.0		12.4%
MULTIPLY					
Reg	0.5	0.3	1.0	0.15	
Direct	0.2	0.3	2.0	0.15	
Immed.	0.1				
Immed. Short	0.1				
Base Rel.	0.5		3.0		8.3%
DIVIDE					
Reg	0.1	0.05	0.1	0.05	
Direct	0.05	0.05	0.2	0.05	
Immed. Short	0.1				
Base Rel	0.05		0.3		1.1%
COMPARE					
Reg	0.7	0.15	0.7	0.15	
Direct	0.7	0.15	0.8	0.15	
Immed.	0.3				
Immed. Short	0.3				
Base Rel	0.7		2.0		6.8%
BRANCH					
Direct	5.0				
IC Rel.	13.0				18.0%
SHIFT					
Reg.	2.0				2.0%
LOGICAL					
Reg.	0.5				
Direct	0.5				
Immed.	1.0				2.0%
BIT					
Reg	1.0				
Direct	1.0				2.0%
TOTALS	74.0%	10.0%	15.0%	1.0%	100%

TABLE III MICROPROCESSOR PERFORMANCE

	CPU Throughput KIPS	Clock MHZ	Memory Access	Memory Cycle
Class I	200	TBD	TBD	TBD
Class II	500	TBD	TBD	TBD

3.2.1.2 External interface. The following paragraphs will describe the performance characteristics of the external interface of the CPU.

3.2.1.2.1 CPU external interface. The detailed block diagram for a CPU is shown in Figure 3. The CPU shall use the following discrete signals to perform in accordance with the MIL-STD-1750 Instruction Set Architecture.

3.2.1.2.1.1 Reset. Activation of the CPU's reset line shall put the CPU in a non-operational state within TBD clock cycles, regardless of its previous state or the state of its other inputs. The CPU shall remain in this state until reset is deactivated. When this occurs, the CPU shall proceed with chip set initialization in accordance with Table VIII. Reset shall be active high in polarity.

3.2.1.2.1.2 CPU clock. The CPU shall utilize this signal to determine the time period of all basic functions. The clock shall tolerate asymmetry of period from 25%/75% to 75%/25%.

3.2.1.2.1.3 Timer clock. The CPU shall utilize a 4 MHz clock to determine the time period of TIMER A and TIMER B. The clocks may be synchronous or asynchronous with the processor clock. The clock shall tolerate asymmetry of period from 25%/75% to 75%/25%.

TABLE VIII

CPU, MMU, BPR, REGISTER/FUNCTION DEFINITION AT INITIALIZATION

Register/Function	Device	Condition
Instruction Counter	CPU	All Zeros
Status Word	"	"
Fault Register	"	"
Pending Interrupt Register	"	"
Interrupt Mask Register	"	"
Interrupt	"	Disabled
DMA Enable	"	"
Timers A&B	"	All Zeros & Counter Pulsed
Trigger GO Reset	"	
Trigger GO Indicator	"	Reset
Normal Power Up	"	Set
Page Registers	MMU	
A2 Field		All Zeros
W Field		"
E Field		"
PPA Field		Logical to Physical
CPU Write Protect Registers	BPR	All Zeros
DMA Write Protect Registers	"	"
Global Memory Protect	"	Enabled

3.2.1.2.1.4 Trigger Go reset. The trigger go reset shall be an active low discrete. The Trigger Go Reset shall be pulsed during CPU initialization and thereafter shall be decoded from the Trigger Go reset command (400BH). The pulse width of the Trigger Go reset shall be TBD.

3.2.1.2.1.5 Faults. The following fault/error discretes shall be recognized. These signals may occur asynchronously, but they are sampled on the trailing edge of Bus busy. The signal levels both low and high shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.5.1 Memory protect error. The memory protect error discrete shall be used by the MMU and the BPR to set bits 0 and 1 of the CPU fault register as defined in paragraph 3.1.3.1.4.

3.2.1.2.1.5.2 Memory parity error. The memory parity error discrete will be used by the external equipment to set bit 2 of the CPU fault register.

3.2.1.2.1.5.3 PIO channel parity error. The PIO channel parity error discrete will be used by the external equipment to set bit 3 of the CPU fault register.

3.2.1.2.1.5.4 DMA channel parity error. The DMA channel parity error discrete will be used by the external equipment to set bit 4 of the CPU fault register.

3.2.1.2.1.5.5 External address error. The external address error discrete will be used by the external eqipment to set bits 5 and 8 of the CPU fault registers as defined in paragraph 6.2.1.

3.2.1.2.1.5.6 PIO channel transmission error. The PIO channel transmission error discrete will be used by the external equipment to set bit 6 of the CPU fault register.

3.2.1.2.1.5.7 Fault bit 7. The fault bit 7 error discrete will be used by the host equipment to set bit 7 of the CPU fault register.

TABLE VI FAULT DETECTION CRITERIA, EXTERNAL

FAULT LOGIC	ILLEGAL INSTRUCTION	EXECUTE PROTECT FAULT	INSTRUCTION PARITY	INSTRUCTION ILLEGAL ADDRESS	ADDRESS STATE FAULT	PRIVILEGED INSTRUCTION	CPU ACCESS FAULT	CPU WRITE PROTECT	CPU DATA ILLEGAL ADDRESS	ILLEGAL IO ADDRESS	CPU DATA PARITY	DMA ACCESS FAULT	DMA WRITE PROTECT	DMA ILLEGAL ADDRESS	DMA MEMORY PARITY	PIO TRANSMISSION	PIO CHANNEL PARITY	DMA CHANNEL PARITY	FAULT BIT 7	BITE	RESERVED – 14	SYSTEM FAULT
	MIL-STD-1750A MACHINE ERROR CONDITIONS																					
• DISCRETES																						
Memory Protect Err *		1					1	1				1	1									
Memory Parity Err *			1								1				1							
PIO Chnl Parity Err																	1					
DMA Chnl Parity Err																		1				
Address Err *			1	1					1	1				1								
PIO Chnl Xmsn Err																1						
Fault Bit 7																			1			
System Fault																						1
• CPU STATUS																						
Inst/Data		1	1	1			0	0	0	0	0	0	0	0	0							
Memory/IO		1	1	1			1	1	1	0	1	1	1	1	1							
DMA Acknowledge		0	0	0			0	0	0	0	1	1	1	1	1							
• DETECTION																						
Internal	■				■	■														■	■	
Sync*		■	■	■			■	■	■	■	■	■	■	■	■							
Asynchronous																■	■	■	■			■
ERROR CLASSIFICATION	UNRECOVERABLE					MAJOR					WARNING											

*SAMPLED BY BUS BUSY

3.2.1.2.1.5.8 System fault. The system fault error discrete will be used by the external equipment to set bit 15 of the CPU fault register.

3.2.1.2.1.6 DMA control. The CPU shall perform the bus acquisition logic function for direct memory access devices in response to the DMA request line if DMA is enabled. After completing its current bus transaction the CPU shall float the memory/IO bus. The CPU memory/IO bus shall remain disabled while DMA request is active or until the end of the current bus transaction as determined by bus busy.

3.2.1.2.1.6.1 DMA request. The CPU shall accept asynchronous bus requests from DMA devices via the DMA request signal. The DMA request signal will be active high when a DMA device is requesting the bus. If enabled, the CPU shall respond to a DMA request at the end of the current CPU address/data bus transaction or within TBD CPU clock cycles.

3.2.1.2.1.6.2 DMA acknowledge. The DMA acknowledge discrete shall be set high when the CPU has disabled its address/data bus in response to a DMA request. DMA acknowledge shall remain high until DMA request is removed.

3.2.1.2.1.6.3 DMA enable. DMA shall be disabled during CPU initialization. Thereafter the response of the CPU to DMA request will be controlled by execution of the DMA enable command (4006H) and the DMA disable command (4007H). The state of the DMA enable function shall be output on the DMA enable output discrete. The DMA enable output discrete shall be active high when the DMA is enabled.

3.2.1.2.1.7 Interrupts. The CPU shall accept an externally generated Power Down Interrupt. In addition the CPU shall provide six general purpose interrupts along with two special purpose interrupts for expanding the number of input/output level interrupts. These signals may be applied asynchronously and are stored in the CPU if the signal pulse duration is greater than TBD nanoseconds. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.7.1 Power down interrupt. The CPU shall accept an externally generated power down interrupt.

3.2.1.2.1.7.2 User interrupts. The CPU shall accept six (6) general purpose interrupts.

3.2.1.2.1.7.3 IOL interrupts. The CPU shall accept two (2) special purpose interrupts for expanding the number of input/output level interrupts.

3.2.1.2.1.8 External request. The CPU shall respond to an external request on a low to high transition of the external request input discrete. The external request shall cause the CPU to suspend operation of the software program at the end of the current instruction cycle. At that time the program counter shall point to the first word of the next instruction to be executed. The CPU shall then execute a test vector command (TBD) to specify the next action to be taken.

3.2.1.2.1.9 Memory/IO bus. The memory/IO bus shall be used to transmit instructions and data to/from the CPU. The CPU shall initiate a new memory/IO bus cycle for every memory reference or XIO command. For XIO commands implemented within the CPU, the command and associated data shall be output on the address/data bus for monitoring by external equipment. For all memory references and XIO commands not implemented within the CPU, bus cycles shall be initiated to transmit/receive the appropriate data from the external equipment.

3.2.1.2.1.9.1 Address/data bus (AD0-AD15). The CPU shall transfer address and data over a 16-bit time multiplexed address/data bus to and from its internal registers. The signal levels both low and high shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.2 Address strobe. Address strobe shall be active for the duration of the address transfer cycle. Address that appear on the memory/IO strobe will be used to latch memory and XIO addresses. Address strobe will also be used to latch address state information from the CPU status bus (reference paragraph 3.2.1.2.1.10). The address strobe shall have a minimum duration of TBD nanoseconds at the maximum processor clock rate. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.3 Ready address. The ready address signal will be driven high asynchronously by a memory module or an I/O device to signify that address will be valid within the time specified for the CPU class. In the absence of ready, the CPU shall extend the address transfer cycle to accommodate slower access time devices. The state of ready shall be ignored if the illegal address discrete is activated or if the address is implemented internally to the CPU. The signal levels both low and high shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.4 Data strobe. The data strobe shall be used to gate data from an external device into the CPU or to control the timing of data being written. On write cycles, data specified setup and hold times shall be observed with respect to data strobe trailing edge. For byte operations which store a byte in memory, the microrocessor shall read a 16-bit word from memory, insert data into the word, and write the new word back into memory on a subsequent data cycle. The data strobe shall have a minimum duration of TBD nanoseconds at the maximum processor clock frequency. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.5 Ready data. The ready data signal will be driven high asynchronously by a memory module or an I/O device to signify that data will be valid within the time specified for the CPU class. In the absence of ready, the CPU shall extend the data transfer cycle to accommodate slower access time devices. The signal levels both low and high shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.6 Direction. This signal shall be used by the external equipment to control the direction of the memory/IO bus buffers. Memory cycles shall be read only or write only with direction valid for the entire duration of the address strobe and the data strobe. For XIO commands implemented within the CPU chip itself direction shall direct data from the CPU to the external equipment regardless of the sense of Bit 15 of the XIO command. Direction shall be active high when the data is to be buffered from the CPU to the external equipment. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.7 Memory/IO. This discrete signal defines whether the current access is for a Memory or I/O address. The Memory/IO signal shall observe the same specified setup and hold times as the address bus lines with respect to address strobe. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.8 Instruction/data. This discrete signal defines whether the current access is for a program instruction or data. The instruction/data signal shall observe the same specified setup and hold times as the address/data bus lines with respect to address strobe (paragraph 3.2.1.2.1.9.2) or data strobe (paragraph 3.2.1.2.1.9.4). The signal levels both low and high shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.9.9 Sync. This signal is active every cycle and may be used for the synchronizing of memory and I/O control. The signal levels both high and low shall conform to the requirements set forth in this specification.

3.2.1.2.1.9.10 Bus busy. This signal may be used to establish the end of a bus cycle. The bus busy signal shall have a minimum duration of TBD nanoseconds at the maximum CPU clock rate. The duration shall be proportionately longer at lower clock rates. The trailing edge of bus busy shall sample bits into the fault register. The signal levels both high and low shall conform to the requirements set forth in this specification.

3.2.1.2.1.10 Status bus (AS/PS). The address state and processor state fields of the CPU status word shall be time multiplexed over the four bit status bus. Address state shall be present on the status word bus during the same time interval as the sixteen bit logical address is present on the memory/IO bus. Processor state shall be present on the status word bus on each data transfer cycle during the data transfer portion of the memory/IO bus cycle.

3.2.1.2.1.11 Normal power up. The normal power up discrete shall be set when the CPU has completed power initialization as required by paragraph 3.2.1.2.1.1. This signal shall remain set until cleared by the Reset Normal Power Up XIO command. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.12 Major error. A major error is the logical OR of the faults indicated by Table VI and occurs if the machine error interrupt is masked by the Interrupt Mask register. Reference Table VI, Fault Detection Criteria, External for a complete illustration

3.2.1.2.1.13 Unrecoverable error. An unrecoverable error is the logical OR of the faults indicated by Table VI and occurs if the machine error interrupt is masked by the Interrupt Mask register. Reference Table VI, Fault Detection Criteria, External for a complete illustration.

3.2.1.2.1.14 Multiprocessor interface. The multiprocessor interface shall contain the following signals for use when implementing bus arbitration in tightly coupled multiprocessor configurations.

3.2.1.2.1.14.1 BUS request. The bus request is an active low signal generated at the beginning of an instruction which requires a bus cycle. The signal levels both high and low shall conform to the requirements set forth in the definitions of this specification.

3.2.1.2.1.14.2 BUS lock. The bus lock shall be an active low open collector output. This signal is used to obtain successive bus cycles where required for special applications. The bus lock should be included in the bus arbitration of multiprocessor systems. The signal levels both high and low shall conform to the requirements set forth in this specification.

3.2.1.2.1.14.3 BUS grant. This signal will be an active high input. Bus grant may be used for multiprocessor operation. A low level shall inhibit address output and halts the processor. The signal levels both high and low shall conform to the requirements set forth in the requirements of this specification.

3.2.1.2.2 Memory management unit (MMU). The MMU shall accept and be compatible with but not be limited by the following interface information from the CPU or BPR.

3.2.1.2.2.1 Memory/IO bus. The Memory/IO bus is received from the CPU and conforms to the requirements of paragraph 3.2.1.2.1.9.

3.2.1.2.2.2 Status bus (AS/PS). The status bus shall be received from the CPU and shall conform to the requirements of paragraph 3.2.1.2.1.10.

3.2.1.2.2.3 Extended address. The extended address interface shall consist of but not be limited by the following signals and groups.

3.2.1.2.2.3.1 Extended address bus (EA0-EA7). The extended address bus shall enable the microprocessor to address up to 1,048,576 words of physical memory. The extended address bus shall consist of the contents (PPA0-PPA7) of the selected page address register shown in Figure 9. The extended address when concatenated with the twelve (12) least significant bits of the address/data bus, compose the physical address. The resulting 20-bit physical address shall address up to 1,048,576 words of physical memory.

3.2.1.2.2.3.2 Ready extended address. The ready extended address signal shall be used to extend the CPU cycle if the system requires additional time to process the address on the address/data bus. Reference paragraph 3.2.1.2.1.9.3.

3.2.1.2.2.4 Reserved interface (R0-R2). The reserved bus shall be connected to the respective reserved bits in the MMU physical page address register.

3.2.1.2.2.5 Memory protect error. The MMU shall generate a protect error when any of the following exist.

(1) The access key does not equal the access lock criteria for the area of memory being addressed. This may be a CPU memory access or a DMA memory access. A memory protect error causes Fault bit 0 (if it is a CPU access) or Fault bit 1 (if it is a DMA access) in the Fault Register to be set. The signal is sampled at the trailing edge of Bus busy.

(2) An attempt has been made to execute a priviledged instruction in an area of memory where the program is not in privileged mode (E bit).

(3) An attempt has been made to write into an area of operand memory which is write protected (W bit).

3.2.1.2.3 Block Protect RAM (BPR). The BPR shall accept and be compatible with but not be limited by the following interface information from the CPU or MMU.

3.2.1.2.3.1 Block protect RAM extended address bus (BA0-BA7). The address lines are described beginning at paragraph paragraph 3.2.1.2.1.9 (CPU) and 3.2.1.2.2.3 (MMU). The BPR address interconnect is shown in Table I.

3.2.1.2.3.2 Memory/IO bus. The memory/IO bus shall consist of the signals described beginning at paragraph 3.2.1.2.1.9.

3.2.1.2.3.3 DMA acknowledge. The DMA acknowledge shall be used to detect memory protect errors when operating under DMA control. The signal shall conform to paragraph 3.2.1.2.1.6.2.

3.2.1.2.3.4 Memory protect enable. This output shall provide an indication that protected memory is enabled.

3.2.1.2.3.5 Memory protect error. This output shall be present before the leading edge of the Data Strobe, and its duration shall be until the end of the memory cycle. An attempt has been made to write into an area of memory which is write protected by the Block Protect RAM. Reference paragraphs beginning at 3.2.1.2.5, and 3.2.1.3.1.1.4.

3.2.1.3 Internal interface.

3.2.1.3.1 CPU internal interface. The CPU internal interface shall be composed of but not limited by the following.

3.2.1.3.1.1 Registers. The registers in the CPU shall perform in accordance with MIL-STD-1750 as follows:

3.2.1.3.1.1.1 Instruction counter. The instruction counter shall be used to access program instructions within a 65,536 word logical instruction address space. The instruction counter relative (ICR) addressing mode allows a short form of memory addressing within a range of -128 to +127 words relative to the current instruction.

3.2.1.3.1.1.2 General registers. The general registers shall be used to access program data within a 65,536 word logical data address space. Memory addressing modes for the general registers shall include the following:

(a) Register direct (R). An addressing mode in which the instruction specified register contains the required operand.

(b) Memory direct (D). An addressing mode in which the instruction contains the memory address of the operand.

(c) Memory direct-indirect (DX). An addressing mode in which the memory address of the required operand is specified by the sum of the content of an index register and the instruction address field. Registers R1..R15 may be specified for indexing.

(d) Memory indirect (I). An addressing mode in which the instruction specified memory address contains the address of the required operand.

(e) Immediate indexed (IMX). An addressing mode in which the sixteen bits if immediate data are summed with the contents of an instruction specified index register. Registers R1..R15 may be specified for indexing.

(f) Base relative (B). An addressing mode in which the content of an instruction specified base register is added to the eight bit displacement field of the sixteen bit instruction. The displacement field is taken to be a positive number in the range of 0..255. The sum points to the memory address of the required operand. Registers R12..R15 may be specified for base registers.

(g) Base relative indexed (BX). The sum of a specified index register and a specified base register is the address of the required operand. Registers R12..R15 may be specified for base registers and R1..R15 may be specified for index registers.

(h) Memory indirect with pre-indexing (IX). An addressing mode in which the sum of the content of a specified index register and the instruction address field is the address of the address the required operand. Registers R1..R15 may be specified for pre-indexing.

(i) Immediate (IM,ISP,ISN). Two forms of immediate data shall be provided. Immediate Long (IM) addressing may be used to specify sixteen bit fixed point data within a range of -32768.. +32767. The short forms of immediate addressing, Immediate Short Positive (ISP) and Immediate Short Negative (SNC), may be used to specifiy fixed point data within the ranges of 1..16 and -16..-1 respectively.

(j) Special (S). The special addressing mode is used where none of the other addressing modes are applicable.

3.2.1.3.1.1.3 Processor status word. The CPU shall use the contents of the status word for controlling program execution, defining the processor state, and interfacing with the MMU.

(a) Condition status. The condition status will allow the programmer to control the program flow for the conditions given in Table V.

TABLE V CONDITION STATUS INTERPRETATION

CS		CONDITION
CPZN	HEX	
0000	0	NOP
0001	1	Less than zero
0010	1	Equal to zero
0011	3	Less than or equal to zero
0100	4	Greater than zero
0100	5	Not equal to zero
0110	6	Greater than or equal to zero
0111	7	Unconditional
1000	8	Carry
1001	9	Carry or less than zero
1010	10	Carry or equal to zero
1011	11	Carry or less than or equal to zero
1100	12	Carry or greater than zero
1101	13	Carry or not equal to zery
1110	14	Carry or greater than or equal to zero
1111	15	Unconditional

(b) Processor state (PS). The processor state shall also determine the legal/illegal criteria for privileged instructions. When PS = 0 and a privileged instruction execution is attempted, the instruction shall be legal and shall be executed properly as defined. When PS ≠ 0 and a privileged instruction execution is attempted, the instruction shall be illegal, shall be aborted, and the privileged instruction fault bit in the fault register (FT)10 shall be set to one.

(c) Address state (AS). For microprocessors which do not contain a MMU, an address state fault shall be generated for any operation which attempts to modify AS to a non-zero value. For implementations which include the extended memory addressing option, AS shall define the group (pair) of page register sets to be used for all instruction and operand references to memory. The new processor status word defined by the interrupt service pointer shall be transferred into the processor status word register provided that the address state code is zero, or an MMU is attached. Otherwise an address state fault shall be generated via bit 11 of the fault register.

(d) Status word commands. The following mandatory commands shall be implemented for use with the processor status register:

(i) Read processor status word (0A00EH). This command shall transfer the contents of the sixteen (16) bit processor status word into register RA. The status word remains unchanged.

(ii) Write processor status word (200EH). This command shall transfer the contents of register RA into the status word provided that the address state code is zero, or that an MMU is attached. Otherwise the contents of the status word shall not be modified, and an address state fault shall be generated via bit 11 of the fault register.

3.2.1.3.1.1.4 Fault register. The fault register shall generate a machine error interrupt whenever any one of its bits is set. The conditions for the setting of each bit shall be as given in Table XVII, Fault Detection Criteria, Internal. The fault register shall be read and cleared by the read and clear fault register command (0A00FH). This command inputs the sixteen (16) bit fault register to register RA. The contents of the fault register are reset to zero. Bit 1 in the pending interrupt register is reset to zero.

3.2.1.3.1.1.5 Pending interrupt register (PI). The pending interrupt request register is software and hardware controlled and contains the pending interrupts that are attempting to vector the instruction counter. A pending interrupt is set by a system interrupt signal. The pending interrupt bit that generates the interrupt request is cleared by hardware action during the interrupt processing prior to initiating software at the address defined by the new IC value. The following mandatory commands shall be implemented for the Pending Interrupt Register:

(a) Clear interrupt request (2001H). All interrupts are cleared (i.e., the pending interrupt register is cleared to all zeros) and the contents of the fault register are reset to zero.

(b) Reset pending interrupt (2004H). The individual interrupt bit to be reset shall be designated in register RA as a right justified four bit code. (OH represents interrupt number 0, 0FH represents interrupt number 15). If interrupt 1H is to be cleared, then the contents of the fault register shall also be set to zero.

(c) Set pending interrupt register (2005H). This command replaces the 16-bit contents of RA with the pending interrupt register with the logical "OR" of the contents of RA and the old pending interrupt register.

If there is a one in the corresponding bit position of the interrupt mask, (same bit set in both the PI and the MK), and the interrupts are enabled, then an interrupt shall occur after execution of the next instruction. If (PI)5 is set to 1, then N is assumed to be 0 in accordance with MIL-STD-1750.

TABLE XVII FAULT DETECTION CRITERIA, INTERNAL

FAULT REGISTER*	BIT No.	MIL-STD-1750A MACHINE ERROR CONDITIONS: ILLEGAL INSTRUCTION	EXECUTE PROTECT FAULT	INSTRUCTION PARITY	INSTRUCTION ILLEGAL ADDRESS	ADDRESS STATE FAULT	PRIVILEGED INSTRUCTION	CPU ACCESS FAULT	CPU WRITE PROTECT	CPU DATA ILLEGAL ADDRESS	ILLEGAL IO ADDRESS	CPU DATA PARITY	DMA ACCESS FAULT	DMA WRITE PROTECT	DMA ILLEGAL ADDRESS	DMA MEMORY PARITY	PIO TRANSMISSION	PIO CHANNEL PARITY	DMA CHANNEL PARITY	FAULT BIT 7	BITE	RESERVED - 14	SYSTEM FAULT
CPU PROTECT	0		●					●	●														
DMA PROTECT	1												●	●									
MEMORY PARITY	2			●								●				●							
PIO CHANNEL PARITY	3																	●					
DMA CHANNEL PARITY	4																		●				
ILLEGAL IO ADDRESS	5										●												
PIO TRANSMISSION	6																●						
SPARE	7																			●			
ILLEGAL MEMORY ADDRESS	8				●					●					●								
ILLEGAL INSTRUCTION	9	●																					
PRIVILEGED INSTRUCTION	10						●																
ADDRESS STATE	11					●																	
RESERVED	12																						
BITE	13																				●	●	●
RESERVED	14																					●	
RESERVED	15																						●
ERROR CLASSIFICATION		UNRECOVERABLE					MAJOR				WARNING												

*MACHINE ERROR INTERRUPT MAY BE MASKED

(d) Read pending interrupt register (0A004H). This command transfers the contents of the pending interrupt register into RA. The pending interrupt register is not altered.

3.2.1.3.1.1.6 Interrupt mask (MK). The interrupt mask register is software controlled and contains a mask bit for each of the system interrupts. The interrupt system is defined in 3.2.1.3.1.2. The following mandatory commands shall be implemented for the Interrupt Mask register:

(a) Set interrupt mask (2000H). This command outputs the 16-bit contents of register RA to the interrupt mask register. A "1" in the corresponding bit position allows the interrupt to occur and a "0" prevents the interrupt from occurring except for those interrupts that are defined such that they cannot be masked.

(b) Read interrupt mask (0A00H). The current interrupt mask is transferred into register RA. The interrupt mask is not altered.

3.2.1.3.1.1.7 Interval timers. The two interval timers shall be software controlled and interfaced with the interrupt system as defined in paragraph 3.1.3.1.8.

At system reset or power up, the timers are initialized to 0000H by the software, then an interrupt request shall be generated after 65,536 counts. A sample of 16-bit counting sequence (shown in hex) is 0000H, 0001H, ..., 7FFFH, 8000H, ..., 0FFFFH, 0000H.

The commands given in Table VII shall be implemented for each timer as described below:

TABLE VII TIMER COMMANDS

	START	HALT	OUTPUT	INPUT
Timer A	4008H	4009H	400AH	0C000H
Timer B	400CH	400DH	400EH	0C00EH

(a) Timer start. This command starts the timer from its current state. The timer is incremented every clock cycle.

(b) Timer halt. This command halts the timer at its current state.

(c) Output timer. The contents of register RA are loaded (i.e., jam transferred) into the timer and the timer automatically starts operation by incrementing from the loaded count in steps of ten microseconds.

(d) Input timer. This command inputs the 16-bit contents of the timer into register RA.

3.2.1.3.1.2 Interrupts. The CPU shall support sixteen (16) interrupt levels as shown in Table II. An interrupt request may occur at any time; however, the interrupt processing must wait until the current instruction is completed. An exception to this is Move Multiple Word which may be interrupted after each single word transfer.

3.2.1.3.1.2.1 Interrupt acceptance. The interrupt system shall have the capability to accept external and internal interrupts. Figure 7 indicates the relationship between the interrupt signals, the pending interrupt register, the interrupt mask register, the priority control logic, the software controllable/accessible signals and the fundamental communications between the interrupt system and the CPU.

3.2.1.3.1.2.2 Interrupt software control. Software shall be able to input from or output to the interrupt mask register as well as the pending interrupt register. Also, software shall be able to disallow recognition of interrupts via the "disable interrupts" signal (without inhibiting interrupt acceptance into the pending interrupt register) and to allow recognition of interrupts via the "enable interrupts" signal.

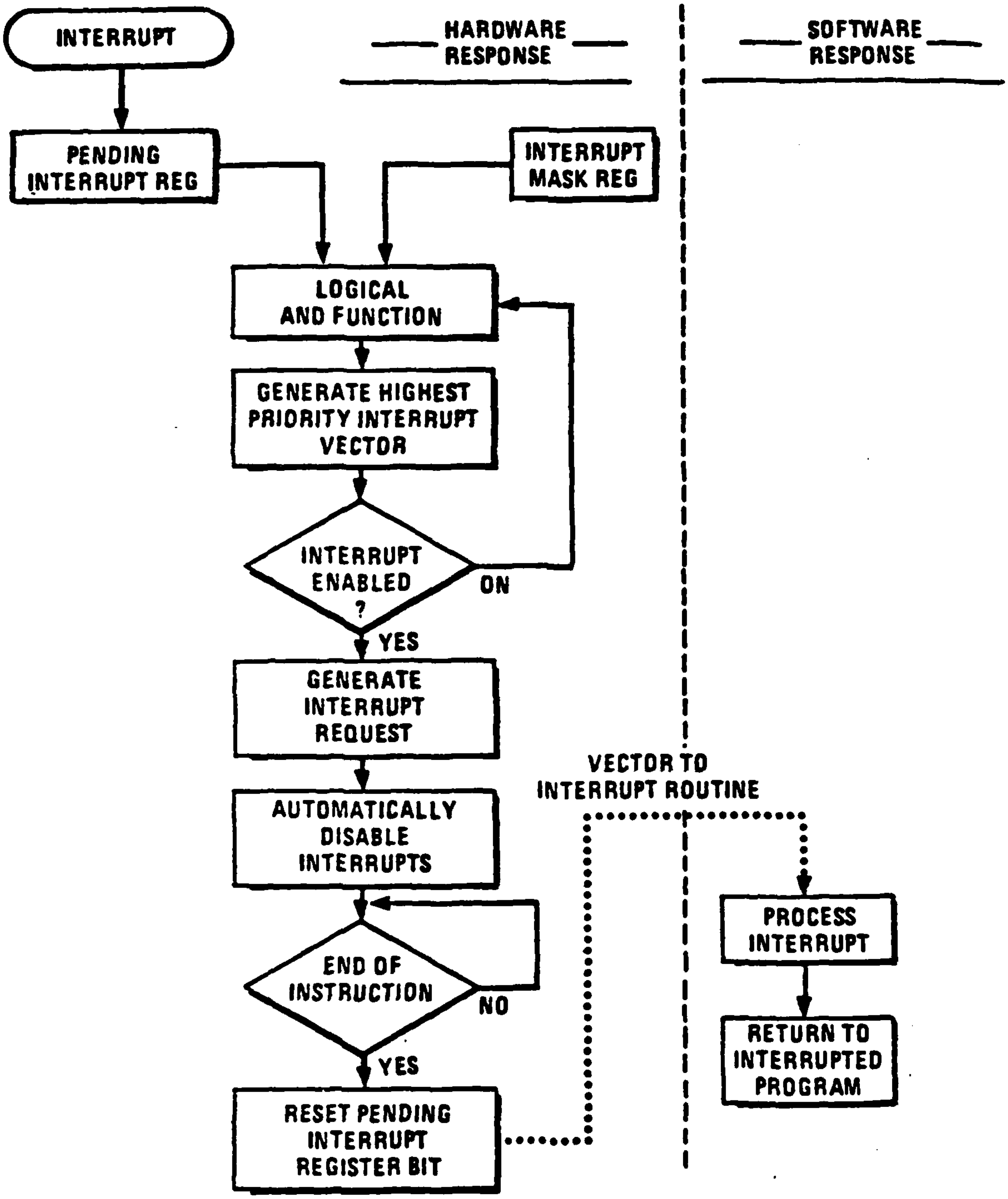


Figure 7 Interrupt System Flowchart

Interrupts shall be controlled via the following mandatory XIO commands:

(a) Enable interrupts (2002H). This command enables all interrupts which are not masked out. The enable operation takes place after execution of the next instruction. If previously disabled, the CPU's interrupt service hardware shall continue to "disable interrupts" for one instruction after the Enable Interrupts instruction has been completed.

(b) Disable interrupts (2003H). This command disables all interrupts (except those that are defined such that they cannot be disabled) at the beginning of the execution of the disable instruction.

3.2.1.3.1.2.3 Interrupt vectoring. The vectoring mechanism shall be as illustrated in Figure 8. For each interrupt there shall be two fixed memory locations in the "vector table:" (1) the first memory location (Linkage Pointer) shall be defined as the address of where to store the current (old) state of the computer (i.e., "old interrupt mask", "old status word", and "old instruction counter"): and (2) the second memory location (Service Pointer shall be defined as the address of the next (new) state of the computer (i.e., "new interrupt mask", "new status word", and "new instruction counter"). Return from interrupts may be accomplished by executing the Load Status (LST/LSTI) instruction with the value/address of the Linkage Pointer for an address field.

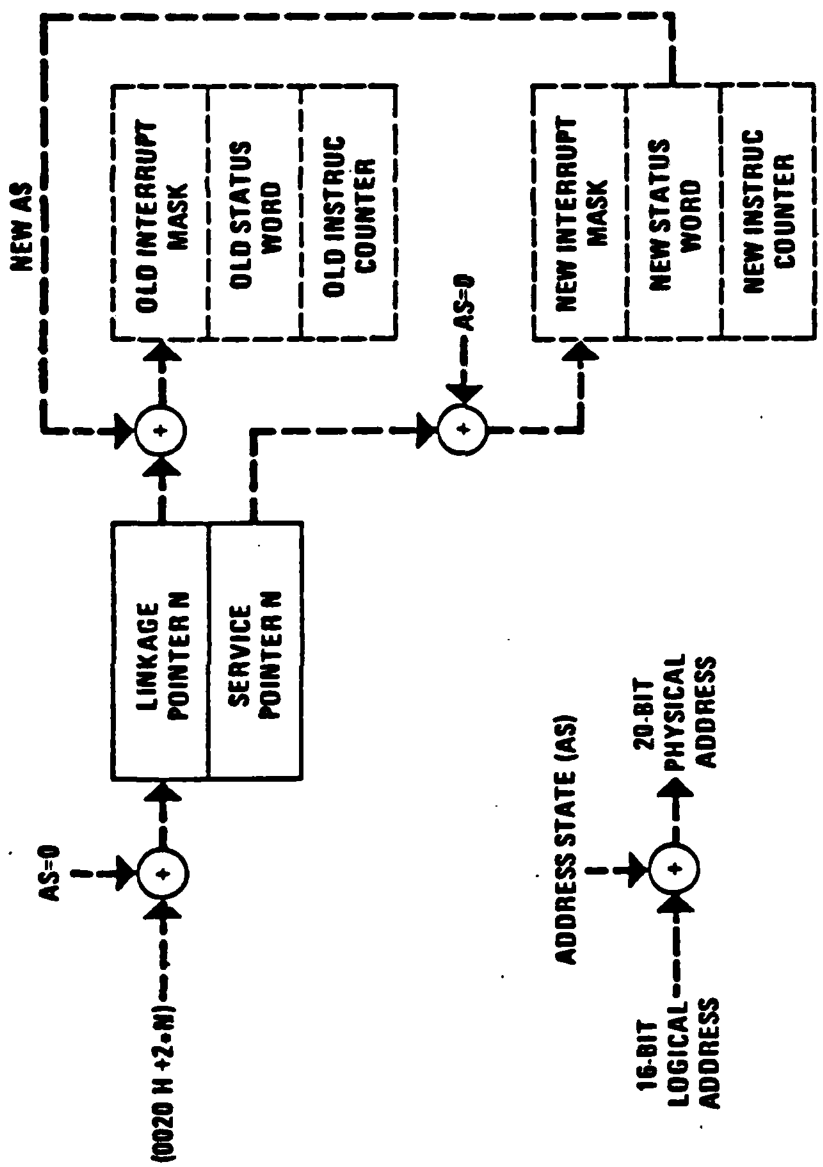


Figure 8 Interrupt Vectoring System

3.2.1.3.2 MMU registers. The MMU shall provide 512 registers in the page file (In XIO memory space). These shall functionally be partitioned into 16 groups with 2 sets per group and 16 page registers per set. Within a group, one set shall be designated for instruction references and the other set for operand references. The page size shall be 4096 words such that one set of 16 page registers shall be capable of mapping 65,536 words defined by a 16-bit logical address.

The Memory Management Unit Block Diagram shown in Figure 9 illustrates the relationship between the MMU page registers, access protect and other system components.

3.2.1.3.2.1 Page register selection. The address state (AS) and instruction/data discretes may be supplied by either the processor or the DMA channel. The address state along with the instruction/data line shall select a group of sixteen page registers to be used for memory mapping. The most significant four bits of the logical address shall be used to establish the particular page register within the selected group to be used for the memory management junctions.

3.2.1.3.2.2 Memory access qualification. The processor status (PS) may be supplied by either the processor status word or the DMA channel. For each of the possible sixteen (16) values of AL code, access shall be acceptable/unacceptable according to Table IX.

References supplying an unacceptable access key code shall not modify any memory location or general registers and an access fault shall be generated. An access fault resulting from a CPU reference attempt shall set fault register bit 0 to cause a machine error interrupt. An access fault resulting from a DMA attempt shall set fault register bit 1 to cause a machine error interrupt.

3.2.1.3.2.3 Memory protection. Memory protection shall be based on the E/W bit of the page registers.

3.2.1.3.2.3.1 Instruction references. When E = 1, an instruction read reference designating that associated page register shall be terminated and an execute protect fault shall be generated. An execute protect fault shall set fault regiater bit 0 to cause a machine error interrupt.

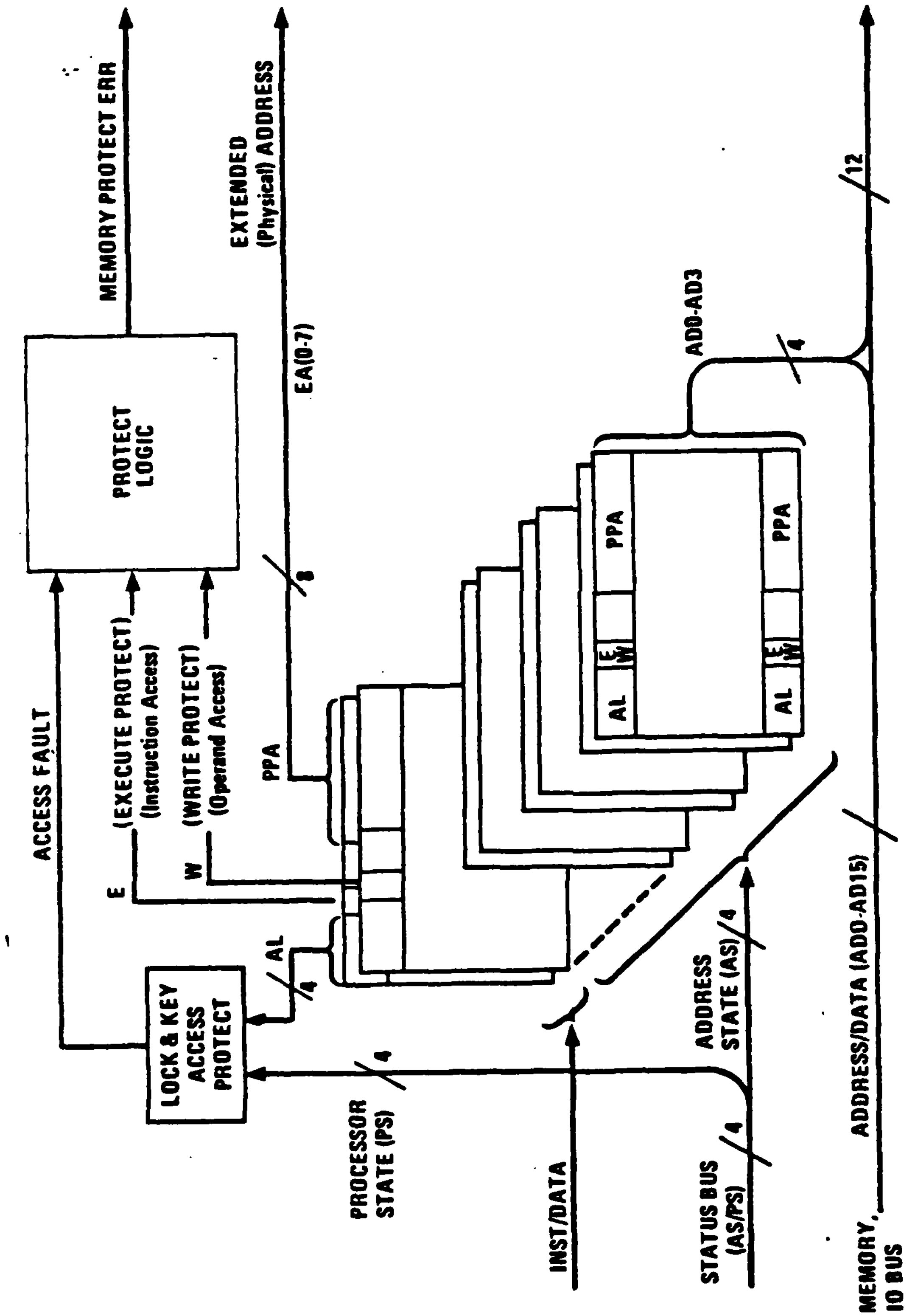


Figure 9 Memory Management Unit Block Diagram

TABLE IX AL CODE TO ACCESS KEY MAPPING

AL CODE	ACCEPTABLE ACCESS KEY CODES
0	0
1	0,1
2	0,2
3	0,3
4	0,4
5	0,5
6	0,6
7	0,7
8	0,8
9	0,9
A	0,A
B	0,B
C	0,C
D	0,D
E	O,E
F	0,1,2,3,4,5,6,7, 8,9,A,B,C,D,E,F

Note that the access lock and key codes defined in the above table have the following characteristics:

a. An access lock code of OFH is an "unlocked" lock code and allows any and all access key codes to be acceptable.

b. An access key code of 0 is a "master" key code and is acceptable to any and all access lock codes.

c. Access key codes 1 through OEH asre acceptable to only their own "matched" lock code or the "unlocked" lock code of OFH.

d. An access key code of OFH is acceptable to only the "unlocked" lock code of OFH.

3.2.1.3.2.3.2 Operand references. When W = 1, any attempted write reference designating that associated page register shall not modify any memory location and a write protect fault shall be generated.

3.2.1.3.2.4 Physical page address. The physical page address (bits 8-15) of each page register shall concentenated with the twelve (12) least significant bits of the logical address to form the twenty (20) bit physical address. This becomes the twenty (20) bit extended memory/IO address at the MMU interface.

3.2.1.3.2.5 MMU commands. The MMU page registers are loaded and read from using the XIO commands given in Table X and described below:

Table X MMU PAGE REGISTER COMMANDS

	Read Commands	Write Commands
Instruction Page Registers	5100H-51FFH	0D100-0D1FF
Operand Page Registers	5200H-52FFH	0D200H-0D2FF

3.2.1.3.2.6 MMU initialization. After the CPU is initialized it initializes the MMU, if the MMU is present to the following state:

TABLE XVIII

Page	Register	Group 0	Enabled
"	"	AL field	All Zeroes
"	"	W field	Zero
"	"	E field	Zero
"	"	PPA field	Exact logical to physical

3.2.1.3.3 Block Protect RAM registers. The following commands shall be available for the Block Protect RAM registers.

3.2.1.3.3.1 Load Memory Protect RAM (5000 + RAM address or 5000 to 5177)$_H$. This command outputs from the CPU the 16 bits of memory protect information to the appropriate RAM register. Each of the 16 bits corresponds to 1024 words of memory. Bit 0 represents the lowest number block and bit 15 represents the highest number block. RAM words 0 thru 63 apply to CPU write protect and words 64 thru 127 apply to DMA write protect.

3.2.1.3.3.2 Read memory protect RAM (0D000H to 0D177H). This command outputs to the CPU the 16 bits of memory protect information from the appropriate RAM register. Each of the 16 bits corresponds to 1024 words of memory. Bit 0 represents the lowest number block ahd bit 15 represents the highest number block. RAM words 0 thru 63 apply to CPU write protect and words 64 thru 127 apply to DMA write protect.

3.2.1.3.3.3 Memory Protect Enable (4003)$_H$. This command allows the memory protect RAM to control memory protection.

3.2.1.3.3.4 Initialization. At power initialization, Block Protect RAM memory shall be initialized by the CPU as follows:

TABLE XIX

Register/Function	Device	Condition
CPU Write Protect Registers	BPR	All Zeroes
DMA Write Protect Registers	BPR	All Zeroes
Global Memory Protect	BPR	Enabled

3.2.1.4 Service conditions - Electrical. Electrical service conditions shall be in accordance with paragraph of MIL-E-5400 except as modified herein.

3.2.1.4.1 Steady-state and transient operation. The Microprocessor shall provide the specified performance with power inputs are within the steady-state limits of Table XI. The microprocessor shall not be damaged for any power excursion within the transient limits of Table XI.

TABLE XI STEADY-STATE AND TRANSIENT VOLTAGE LIMITS

	Steady-State Limit	Transient Limits
PRIMARY POWER	+ 5 VOLTS ± 10%	0 to + 7 VOLTS
SECONDARY POWER	+12 VOLTS ± 10%	0 to +18 VOLTS

3.2.1.4.2 Power requirements. Total power dissipation in the microprocessor shall not exceed the detail limits specified in Table XII.

TABLE XII MICROPROCESSOR POWER DISSIPATION

	CPU	CPU + MMU	Block Protect RAM
Class I	5 Watts	7 Watts	2 Watts
Class II	10 Watts	14 Watts	4 Watts

3.2.1.4.2.1 Signal interface. The microprocessor shall interface with conventional low-power Schottky TTL signal levels as defined herein. Note positive signal excursions are specified with reference to the primary power source (VCC).

3.2.1.4.2.1.1 Input loading. Microprocessor signal current supplied from an input node shall be not greater than 800 microamperes when the input voltage is within the range of -0.6 to +0.8 volts. The node shall sink not more than 40 microamperes whenever the input voltage is in the range of 2.0 to VCC +0.5 volts.

3.2.1.4.2.1.2 Output drive. Microprocessor signal current available at an output node shall not be less than 400 microamperes at an output voltage level of 2.4 volts. The output node shall be capable of sinking 2000 microamperes while maintaining an output voltage of less than 0.5 volts.

3.2.1.4.3 Warm-up. The microprocessor shall conform to these performance requirements with no warm-up time after application of all primary and secondary power under the conditions and within the temperature extremes of 3.2.5.1.1.

3.2.1.4.4 Thermal design. The microprocessor thermal design shall be such that the microprocessor will delivery specified performance with a case temperature of 125°C. This is a design goal. The minimum acceptable result is a microprocessor which will delivery specified performance with a case temperature of 100°C.

3.2.1.4.5 Processor test. The processor shall have a minimum of built-in-test hardware. However, the Processor shall reply on a short built-in-test (BIT) routine which shall perform a Go/No Go self test of the processor and reset the Go/No Go timer. This bit routine shall provide a 90 percent confidence of error detection. The memory requirement for this routine shall not exceed 400 sixteen bit words.

3.2.1.4.6 Built-in functions. The 8-bit primary op code 4FH shall be reserved for implementing built-in functions without redesign of the microprocessor chip set. This requirements shall be met in either of the following ways:

a. If the microprocessor is designed using a microcoded architecture, a minimum of 30% reserve shall be provided for definition of built-in functions after implementation of the MIL-STD-1750 ISA and BIT requirements.

b. Alternatively, the microprocessor chip set shall be capable of being extended through addition of dedicated built-in function processor of new design.

3.2.2 Physical characteristics. The physical characteristics of the microprocessors shall reflect consideration of the human factors and environmental factors during all phases of equipment operation and maintenance.

3.2.2.1 Weight. The weight of the microprocessor shall not exceed the detail values given in Table XIII.

TABLE XIII. MICROPROCESSOR WEIGHT LIMITS

	CPU	CPU + MMU	Block Protect RAM
Class I	TBD	TBD	TBD
Class II	TBD	TBD	TBD

3.2.2.2 Size. The microprocessor chip set when manufactured in a form compatible with packaging with dual in line packaged components shall conform to the area and height restrictions contained herein.

3.2.2.2.1 Area. The microprocessor chip set shall be capable of being packaged within a board area with the dimensions given in Table XIV.

TABLE XIV

MICROPROCESSOR PACKAGING AREAS

	CPU	CPU + MMU	Block Protect RAM
Area (LXW)	16in^2	20in^2	4in^2

3.2.2.2.2 Height. The microprocessor components shall be capable of being mounted as shown in Figure 10.

3.2.3 Reliability. The equipment shall provide a mean time between failure (MTBF) of no less than 10,000 hours when operated within the environmental extremes required by this specification.

3.2.4 Maintainability. Equipment maintenance provisions shall conform to the requirements of this specifiation, and the equipment shall meet the following quantitative requirements.

3.2.4.1 Microprocessor testability. Microprocessor maintenance shall consist of all maintenance operations necessary for complete test, repair and/or overhaul of the microprocessors. To facilitate these maintenance operations utilizing Automatic Test Equipment (ATE), each microprocessor shall be designed to meet the following fault isolation capability.

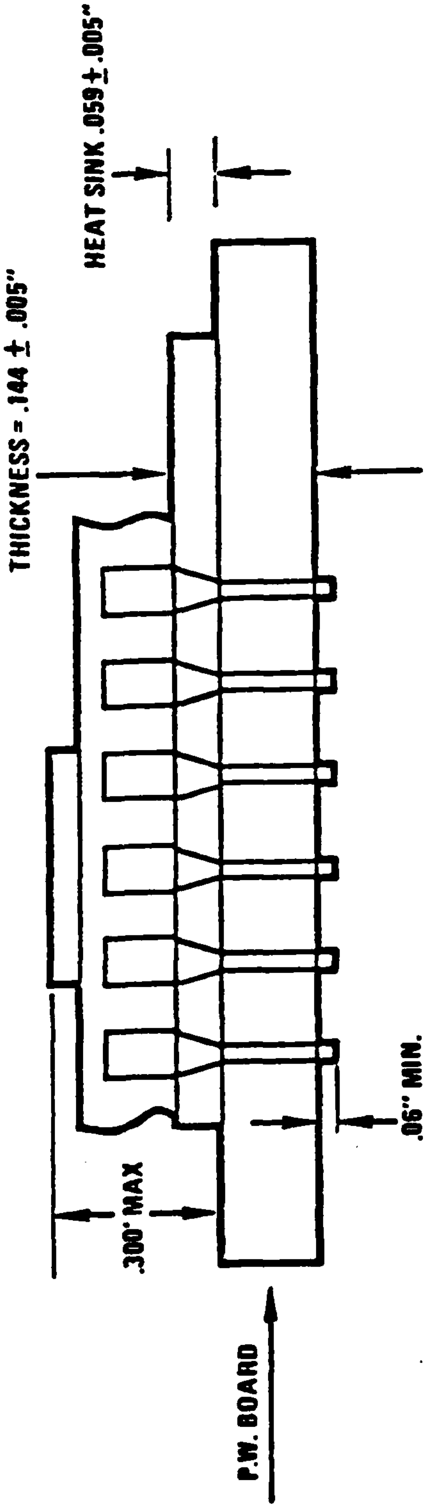


Figure 10 Microprocessor Component Mounting

a. The level of fault detection shall be 97% of all possible functional failures.

b. Fault isolation to a single circuit element (component) in 85% of the detected faults.

c. Fault isolation to two circuit elements in 95% of the detected faults.

d. Fault isolation to three circuit elements in 99% of the detected faults.

The minimum acceptable level of fault detection shall be detection of 85% of the faults which cause a "stuck at one" or "stuck at zero" logic condition on at least one interface signal for each chip in the chip set.

3.2.4.2 Testability verification. Test patterns, test grading (85% detection), and a list of untested nodes shall be developed for each chip in the microprocessor chip set.

3.2.4.3 Testability models. N/A

3.2.4.4 Maintenance diagnostic. The maintenance diagnostic shall be in two parts. Part 1 will be a short program for integrating into operational software. This test is intended to give a quick verification of the proper operating integrity of the machine. Part 2 will be a detailed off-line test package to be used for maintenance of the processor. This test will rigorously exercise the components to enable detection and isolation of component malfunctions. The test program shall progress from simple tests to complex and severe tests. The test shall also progress from global testing to specific isolation type tests with tests calling lower level tests in a logical sequence. Each test shall be able to operate independently of other tests. Isolation to a particular chip in the chip set is desired if possible.

3.2.5 Environmental conditions. All equipment shall deliver specified performance when subjected to any applicable combination of environmental conditions specified in MIL-E-5400 except as modified in this specification.

3.2.5.1 Temperature and altitude environments. The equipment shall deliver specified performance when subjected to any applicable combination of the following conditions, except as otherwise noted herein.

3.2.5.1.1 Surrounding air temperature.

a. Storage - The equipment shall withstand extended exposure to surrounding air temperatures within the range of minus 65°C to plus 150°C while in dead storage, either "on the shelf" or installed in nonoperational aircraft.

b. Operational - The equipment shall withstand extended nonoperational exposure as well as deliver specified performance on the bench and when installed in operational equipment and subjected to surrounding air temperatures of minus 45C to plus 95C.

c. Transient Characteristics - The temperature of the air surrounding any unit of equipment may vary at a rate as high as 1°C per second within the applicable range.

3.2.5.1.2 Surrounding air pressure. Unless otherwise specified, the equipment shall withstand extended nonoperating exposure as well as deliver specified performance when subjected to surrounding air pressure ranging from 15.5 to 0.44 psia. Operation at pressures from 1.04 to 0.44 psia will be limited to two minutes per exposure. The air pressure surrounding the equipment may be assumed equal to the freestream static pressure. The surrounding air pressure may vary at a rate as high as 0.6 psi per second.

3.2.5.2 Explosive atmosphere. Explosive atmosphere, MIL-E-5400, paragraph 3.2.24.10, is applicable.

3.2.5.3 Humidity and moisture. The equipment shall meet the requirements of MIL-E-5400, pararaph 3.2.24.4.

The following design techniques shall be employed to preclude moisture induced problems:

a. Maximum use of hermetically sealed components

b. Use of conformal coatings on all electrical cords

c. Use of cold plate techniques for heat dissipation

d. Use of sealed connectors both internally and externally on electronic assemblies

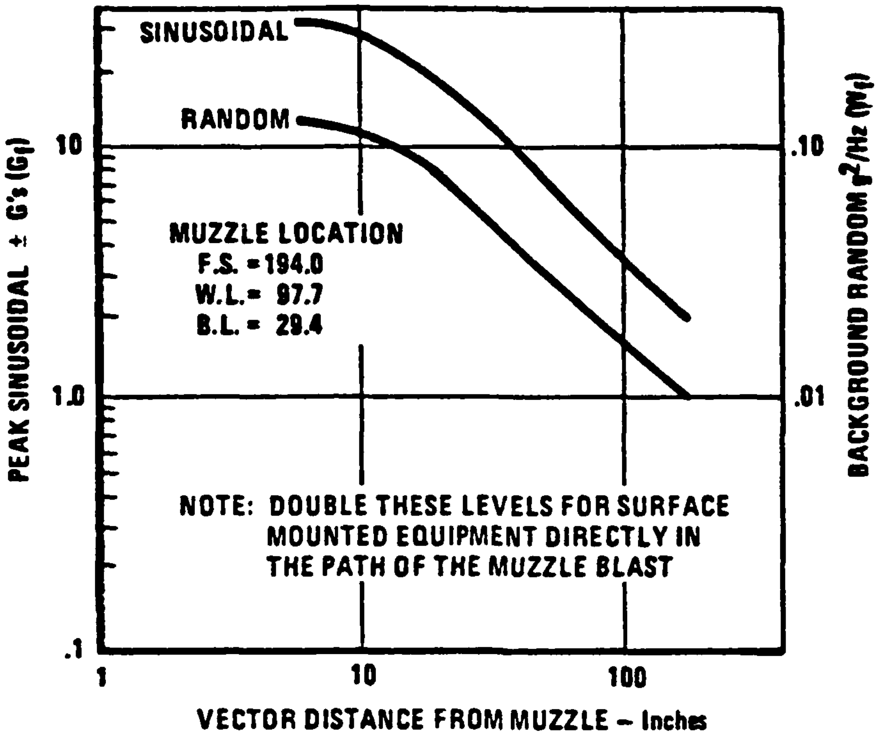
e. Arrangement of internal components to preclude localized entrapment of moisture.

3.2.5.4 Salt-sea atmosphere. Salt-sea atmosphere conditions of MIL-E-5400, paragraph 3.2.24.9, are applicable.

3.2.5.5 Fungus. Fungus conditions of MIL-E-5400, paragraph 3.2.24.8, are applicable.

3.2.5.6 Sand and dust. Sand and dust conditions of MIL-E-5400, paragraph 3.2.24.7, are applicable.

3.2.5.7 Vibration environment. Equipment will be designed to operate satisfactorily in the vibratory environment for the life of the F-16. The primary vibration environment is random in nature and is caused by the engine exhaust noise on the ground and by turbulent air flow in-flight. Other sources of vibration are associated with engine out-of-balance and gunfiring. Sinusoidal vibration is the predominat excitation for engine mounted equipment and for components when exposed to gunfiring. Specific requirements for a given item may deviate from either the methods or levels contained herein if information is available to justify the change. The vibration environment for the microprocessor is shown in Figure 12. The



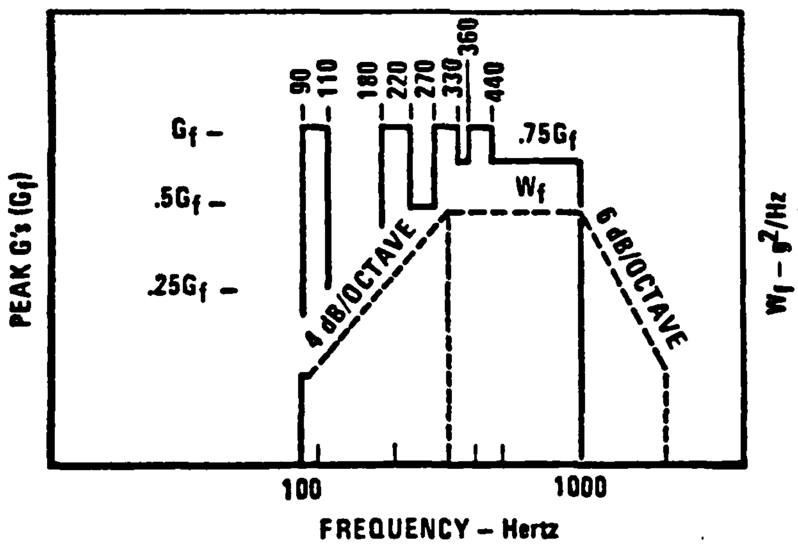


Figure 12 Combined Random and Sinusoidal Vibration Environment

environment consists of a sinusoidal excitation superimposed on a background random vibration spectrum. The intensity of the vibration is a function of the distance from the gun muzzle to the centroid of the equipment.

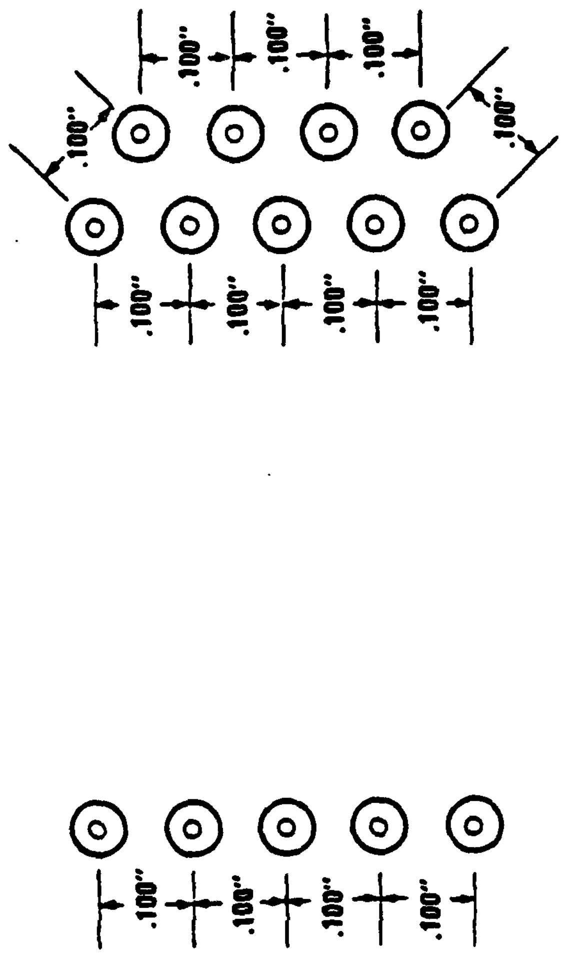
3.2.5.8 Shock. The microprocessor shall comply with the requirements of MIL-E-5400, paragraph 3.2.24.6.1, except that the waveform and amplitude shall be as specified in Figure 13.

3.2.6 Transportability. The design of the microprocessor shall permit and facilitate all modes of transportation anticipated for employment and logistics support. Transportability characteristics shall receive an appropriate balance of priorities in relation to other design characteristics.

3.3 Design and construction. The design and construction of the equipment shall be in accordance with Class 2X of MIL-E-5400. The Design layout and assembly of component parts shall be such as to facilitate quantity production and to result in minimum size and weight. Maximum consideration shall be given to standardization, reliability, maintainability, EMI/EMC, survivability, safety and logistics to obtain the service performance of the equipment. Components shall be packaged on dual in-line plug-in ceramic substrates. The seller shall define the dual in-line size and pin out at the time of Critical Design Review (CDR).

3.3.1 Materials, processes, and parts. The materials, processes and parts shall be in accordance with MIL-E-5400.

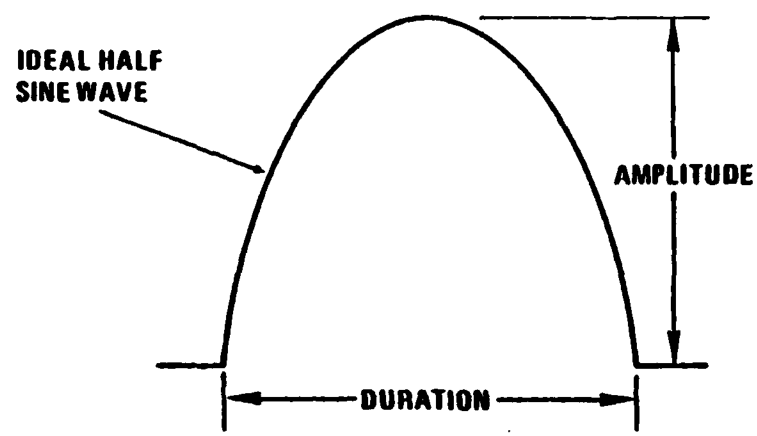
3.3.1.1 Design layout. The dual in-line plug-in ceramic substrates shall be designed for compatibility with standard pin spacings of 0-100". Higher density packaging may be used if pins are offset so that the pin to pin spacing of 0.100" minimum is maintained. Pin spacings on a PWB are illustrated in Figure 11.



b) High Density Pin Spacing

a) Standard Pin Spacing

Figure 11 PWB Design Layout Minimum Lead Spacing



	PEAK VALUE G's	NOMINAL DURATION MILLISECONDS
BASIC DESIGN	15	11

Figure 13 Shock Requirements

3.3.1.2 Microcircuits. Microcircuits shall conform to Requirement 64 of MIL-STD-454, Class B devices. Nonstandard microcircuits shall be screened to Method 5004.2, MIL-STD-883, Class B requirements as a minimum. Plastic encapsulated microcircuits shall not be used.

3.3.1.3 Semiconductors. Semiconductors shall conform to Requirement 30 of MIL-STD-454, JANTX requirements. Plastic encapsulated semiconductors shall not be used.

3.3.1.4 Passive devices. Passive devices shall be selected from the Established Reliability (ER) specifications of MIL-STD-454.

3.3.1.5 Standard parts. Parts definitions are shown in MIL-STD-891. All parts included in Federal Stock Classes (FSC) defined in MIL-STD-891 used in the design shall be selected from the Program Parts Selection List (PPSL), 16PP027. All parts not in 16PP027 are considered unapproved.

3.3.1.6 Nonstandard parts. Nonstandard parts require Parts Control Board approval prior to use on a production program. If the parts listed in 16PP027 are not adequate for the design, in the microprocessor chip set, the Seller shall prepare data supporting a request for approval of a nonstandard part to General Dynamics.

3.3.2 Electromagnetic interference and compatibility. The generation of and susceptibility to electromagnetic interference shall be controlled in the microprocessor. The integrated circuits that are members of the subject microprocessor chip set shall be capable of withstanding an electrostatic discharge test preformed in accordance with DOD-STD-1686, Appendix B. A 1000 V signal is recommended to be used in the circuit specified.

3.3.3 Product marking. The product shall be identified by permanent-type marking applied directly to the part using any suitable means authorized by MIL-STD-130. The priority as listed below is preferred but not required.

a. Design Activity Code Ident and Part Number
b. Serial Number
c. Space for National Stock Number
d. Contract Number
e. Special Characteristics

3.3.4 Workmanship. Workmanship shall conform to MIL-STD-454, Requirement 9.

3.3.5 Interchangeability. Interchangeability shall exist between all units and replaceable assemblies, subassemblies, and parts for all equipment delivered on the contract in accordance with MIL-STD-454, Requirement 7.

3.3.6 Safety criteria. The microprocessor shall incorporate design features which will promote the safety of those personnel who will be employed in the system.

3.3.6.1 Toxicity. Personnel exposure to toxic air contaminants during microprocessor operation, maintenance, and training shall not exceed the ceiling values of OSHA Standard 1910.93.

3.3.7 Human engineering. The design and construction of the equipment shall comply with the applicable requirements of MIL-STD-1472 and AFSC DF 1-3.

3.3.8 Switching transients. Transients from switching within the equipment, whether automatically or manually controlled, shall be minimized by good equipment design.

3.3.9 Overload protection. In addition to the overload protection requirements of paragraph 3.2.20 of MIL-E-5400, the equipment shall be protected from chain reaction failures including those from external overloads (shorts) caused by grounding of external wiring during installation, test, or other causes. Insofar as practical, no damage to a microprocessor shall result from open circuit or grounding of wiring external to the microprocessor.

3.3.10 Thermal analysis. A thermal analysis shall be conducted on the design of the equipment in order to determine the optimum method of dissipating heat. The analysis shall be electrical components/circuits on the cards so that the heat dissipated by these components/circuits may have a path as short or direct as possible to the outside case. This analysis shall document the alternate approaches which were considered and shall present the rationale used to arrive at the final selection.

3.4 Documentation. The following documentation shall be applicable:

a. Drawings

b. Test, plans, procedures, and results/reports.

3.5 Logistics. The microprocessor shall be designed with a goal of minimizing the costs of operation and maintenance and to enchance the accomplishment of required maintenance.

3.5.1 Maintenance. The policy of three levels of maintenance (organization, field, and depot) will be employed for support of the equipment. The concept requires repaid accessibility to all LRUs designed for fast removal and replacement, and onboard capability for equipment checkout and isolation. The intermediate level maintenance will consist primarily of repairing LRUs returned from the flight line. This will be accomplished by removing and replacing shop replaceable assemblies, subassemblies, or units. The equipment shall be designed for, and shall be equipped with, connections at accessible locations for test equipmenty as may be required to perform checkout of the equipment without requiring its removal. Internal and external test points shall be incorporated into the equipment connection of the required test equipment during bench testing, calibration, and troubleshooting.

3.5.2 Supply. The microprocessor chip set design shall reflect the following supply considerations.

a. Requirements shall favor maximum use of standard (common usage) parts and components unless special/peculiar items are shown to be more cost effective.

b. Modular design shall be used so as to optimize demands on the supply system.

3.6 Precedence. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered as superseding requirements. However, in the event of a conflict between MIL-STD-1750 and this document, the contents of MIL-STD-1750 shall be considered to be a superseding requirement.

4. QUALITY ASSURANCE PROVISION

4.1 General. Unless otherwise specified, no adjustment, repairs or maintenance shall be allowed during tests. The supplier shall obtain General Dynamics approval of test procedures formulated and based on this specification prior to conducting any tests. When a functional test is conducted, a record shall be made of all test data necessary to determine compliance with the performance requirements. Test results shall form the basis of acceptance. When the quality from a lot designated by the specified plan fails to meet the specification, acceptance of all items of the lot shall be withheld until the extent and cause of failure is determined. When authorized by General Dynamics to make corrections and after such corrections are made, all necessary tests shall be conducted. Test failures that occur during environmental testing shall be documented, coordinated, and resolved via the Test Failure and Resolution Report (TFRR).

4.1.1 Responsibility for tests. Unless otherwise specified, the supplier shall be responsible for the performance of all inspection and test requirements as specified herein. The supplier may use his own or any commercial laboratory acceptable to General Dynamics. General Dynamics reserves the right to perform any of the inpsections or tests set forthwhere such inspections or tests are deemed necessary to assure that the item conforms to the specified requirements and to witness or monitor any or all tests at either the suppliers facility or subcontracted facilities.

4.1.2 Verification cross reference index (VCRI). A VCRI is included in Table XV and provides for accountability for each Section 3 requirement, corresponding Section 4 verification requirment, and method of verification.

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable   2 = Analysis   4 = Test
1 = Inspection   3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3. Requirements	X					
3.1 Item definition	X					
3.1.1 Functional diagram			X			4.1.4(2)
3.1.2 External interface			X			4.1.4(2)
3.1.2.1 CPU external interface	X					
3.1.2.1.1 Reset			X			4.1.4(2)
3.1.2.1.2 CPU clock			X			4.1.4(2)
3.1.2.1.3 Timer clock			X			4.1.4(2)
3.1.2.1.4 Trigger go reset			X			4.1.4(2)
3.1.2.1.5 Faults			X			4.1. (2)
3.1.2.1.6 DMA control			X			4.1.4(2)
3.1.2.1.6.1 DMA request			X			4.1.4(2)
3.1.2.1.6.2 DMA acknowledge			X			4.1.4(2)
3.1.2.1.6.3 DMA enable			X			4.1.4(2)
3.1.2.1.7 Interrupts			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.1.2.1.8 External request			X			4.1.4(2)
3.1.2.1.9 Memory/IO bus			X			4.1.4(2)
3.1.2.1.9.1 Address/data bus (AD0-AD15)			X			4.1.4(2)
3.1.2.1.9.2 Address strobe			X			4.1.4(2)
3.1.2.1.9.3 Ready address			X			4.1.4(2)
3.1.2.1.9.4 Data strobe			X			4.1.4(2)
3.1.2.1.9.5 Ready data			X			4.1.4(2)
3.1.2.1.9.6 Direction			X			4.1.4(2)
3.1.2.1.9.7 Memory/IO			X			4.1.4(2)
3.1.2.1.9.8 Instruction/ data			X			4.1.4(2)
3.1.2.1.9.9 Sync			X			4.1.4(2)
3.1.2.1.9.10 Bus busy			X			4.1.4(2)
3.1.2.1.10 Status word bus (AS/PS)			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.1.2.1.11 Normal power up			X			4.1.4(2)
3.1.2.1.12 Major error			X			4.1.4(2)
3.1.2.1.13 Unrecoverable error			X			4.1.4(2)
3.1.2.1.14 Multiprocessor interface			X			4.1.4(2)
3.1.2.1.14.1 Bus request			X			4.1.4(2)
3.1.2.1.14.2 Bus lock			X			4.1.4(2)
3.1.2.1.14.3 Bus grant			X			4.1.4(2)
3.1.2.2 Memory management Unit	X					
3.1.2.2.1 Memory/IO bus			X			4.1.4(2)
3.1.2.2.2 Status bus (AS/PS)			X			4.1.4(2)
3.1.2.2.3 Extended address	X					
3.1.2.2.3.1 Extended address bus (EA0-EA7)			X			4.1.4(2)
3.1.2.2.3.2 Ready extended address			X			4.1.4(2)
3.1.2.2.4 Reserved interface (R0-R2)			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable   2 = Analysis   4 = Test
1 = Inspection   3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.1.2.2.5 Memory protect error			X			4.1.4(2)
3.1.2.3 Block Protect RAM	X					
3.1.2.3.1 Block Protect RAM Extended Address bus (BA0-BA7)			X			4.1.4(2)
3.1.2.3.2 Memory/IO bus			X			4.1.4(2)
3.1.2.3.3 DMA acknowledge			X			4.1.4(2)
3.1.2.3.4 Memory protect enable			X			4.1.4(2)
3.1.2.3.5 Memory protect error			X			4.1.4(2)
3.1.3 Internal interface	X					
3.1.3.1 CPU internal interface	X					
3.1.3.1.1 Instruction counter			X			4.1.4(2)
3.1.3.1.1.2 General registers			X			4.1.4(2)
3.1.3.1.1.3 Processor status word			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable   2 = Analysis   4 = Test
1 = Inspection   3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.1.3.1.3.1 Condition status			X			4.1.4(2)
3.1.3.1.3.2 Reserved bits			X			4.1.4(2)
3.1.3.1.3.3 Processor state (PS)			X			4.1.4(2)
3.1.3.1.3.4 Address state (AS)			X			4.1.4(2)
3.1.3.1.4 Fault register (FT)			X			4.1.4(2)
3.1.3.1.5 Pending interrupt register (PI)			X			4.1.4(2)
3.1.3.1.6 Interrupt mask (IM)			X			4.1.4(2)
3.1.3.1.7 Interval timers			X			4.1.4(2)
3.1.3.1.8 Interrupt definitions			X			4.1.4(2)
3.1.3.1.8.1 Machine error			X			4.1.4(2)
3.1.3.1.8.2 Arithmetic exceptions			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.1.3.1.8.3 Executive call			X			4.1.4(2)
3.1.3.1.8.4 Power down			X			4.1.4(2)
3.1.3.1.8.5 General purpose interrupts (User 0-5)			X			4.1.4(2)
3.1.3.1.8.6 Input/output level interrupts			X			4.1.4(2)
3.1.3.2 Memory management unit registers			X			4.1.4(2)
3.1.3.3 Block Protect RAM Registers			X			4.1.4(2)
3.2 Characteristics	X					
3.2.1 Performance	X					
3.2.1.1 Throughput					X	4.2.3.3
3.2.1.2 External interface	X					
3.2.1.2.1 CPU external interface	X					
3.2.1.2.1.1 Reset					X	4.2.3.3
3.2.1.2.1.2 CPU clock					X	4.2.3.3

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.1.3 Timer clock					X	4.2.3.3
3.2.1.2.1.4 Trigger go reset					X	4.2.3.3
3.2.1.2.1.5 Faults	X					
3.2.1.2.1.5.1 Memory protect error					X	4.2.3.3
3.2.1.2.1.5.2 Memory parity error					X	4.2.3.3
3.2.1.2.1.5.3 PIO channel parity error					X	4.2.3.3
3.2.1.2.1.5.4 DMA channel parity error					X	4.2.3.3
3.2.1.2.1.5.5 External address error					X	4.2.3.3
3.2.1.2.1.5.6 PIO channel transmission error					X	4.2.3.3
3.2.1.2.1.5.7 Fault bit 7					X	4.2.3.3
3.2.1.2.1.5.8 System fault					X	4.2.3.3
3.2.1.2.1.6 DMA control					X	4.2.3.3
3.2.1.2.1.6.1 DMA request					X	4.2.3.3

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.1.6.2 DMA acknowledge					X	4.2.3.3
3.2.1.2.1.6.3 DMA enable					X	4.2.3.3
3.2.1.2.1.7 Interrupts	X					
3.2.1.2.1.7.1 Power down interrupt					X	4.2.3.3
3.2.1.2.1.7.2 User interrupts					X	4.2.3.3
3.2.1.2.1.7.3 IOL interrupts					X	4.2.3.3
3.2.1.2.1.8 External request					X	4.2.3.3
3.2.1.2.1.9 Memory/IO bus					X	4.2.3.3
3.2.1.2.1.9.1 Address/data bus					X	4.2.3.3
3.2.1.2.1.9.2 Address strobe					X	4.2.3.3
3.2.1.2.1.9.3 Ready address					X	4.2.3.3
3.2.1.2.1.9.4 Data strobe					X	4.2.3.3
3.2.1.2.1.9.5 Ready data					X	4.2.3.3
3.2.1.2.1.9.6 Direction					X	4.2.3.3

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable   2 = Analysis   4 = Test
1 = Inspection   3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.1.9.7 Memory/IO					X	4.2.3.3
3.2.1.2.1.9.8 Instruction/data					X	4.2.3.3
3.2.1.2.1.9.9 Sync					X	4.2.3.3
3.2.1.2.1.9.10 Bus busy					X	4.2.3.3
3.2.1.2.1.10 Status bus (AS/PS)					X	4.2.3.3
3.2.1.2.1.11 Normal power up					X	4.2.3.3
3.2.1.2.1.12 Major error					X	4.2.3.3
3.2.1.2.1.13 Unrecoverable error					X	4.2.3.3
3.2.1.2.1.14 Multuprocessor interface					X	4.2.3.3
3.2.1.2.1.14.1 Bus request					X	4.2.3.3
3.2.1.2.1.14.2 Bus lock					X	4.2.3.3
3.2.1.2.1.14.3 Bus grant					X	4.2.3.3
3.2.1.2.2 Memory management unit (MMU)	X					

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.2.1 Memory/IO bus	X					
3.2.1.2.2.2 Status bus (AS/PS)					X	4.2.3.3
3.2.1.2.2.3 Extended address	X					
3.2.1.2.2.3.1 Extended address bus (EA0-7)					X	4.2.3.3
3.2.1.2.2.3.2 Ready/ extended address					X	4.2.3.3
3.2.1.2.2.4 Reserved interface (R0-R2)					X	4.2.3.3
3.2.1.2.2.5 Memory protect error					X	4.2.3.3
3.2.1.2.3 Block Protect RAM	X					
3.2.1.2.3.1 Extended address(BA0-BA7)					X	4.2.3.3
3.2.1.2.3.2 Memory/IO block protect RAM	X					
3.2.1.2.3.3 DMA acknowledge					X	4.2.3.3

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.3.4 Memory protect enable					X	4.2.3.3
3.2.1.2.3.5 Memory protect error					X	4.2.3.3
3.2.1.3 Internal interface	X					
3.2.1.3.1 CPU	X					
3.2.1.3.1.1 Registers	X					
3.2.1.3.1.1.1 Instruction counter					X	4.2.3.3
3.2.1.3.1.1.2 General registers					X	4.2.3.3
3.2.1.3.1.1.3 Processor status word					X	4.2.3.3
3.2.1.3.1.1.4 Fault register					X	4.2.3.3
3.2.1.3.1.1.5 Pending interrupt register (PI)					X	4.2.3.3
3.2.1.3.1.1.6 Interrupt mask (MK)					X	4.2.3.3
3.2.1.3.1.1.7 Interval timers					X	4.2.3.3

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.3.1.2 Interrupts					X	4.2.3.3
3.2.1.3.1.2.1 Interrupt acceptance					X	4.2.3.3
3.2.1.3.1.2.2 Interrupt software control					X	4.2.3.3
3.2.1.3.1.2.3 Interrupt vectoring	X					
3.2.1.3.2 MMU registers					X	4.2.3.3
3.2.1.3.2.1 Page register selection					X	4.2.3.3
3.2.1.3.2.2 Memory access qualification					X	4.2.3.3
3.2.1.3.2.3 Memory protection					X	4.2.3.3
3.2.1.3.2.3.1 Instruction references					X	4.2.3.3
3.2.1.3.2.3.2 Operand references					X	4.2.3.3
3.2.1.3.2.4 Physical page address					X	4.2.3.3
3.2.1.3.2.5 MMU commands	X					

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.3.2.6 MMU initialization					X	4.2.3.3
3.2.1.3.3 Block protect RAM registers	X					
3.2.1.3.3.1 Load memory protect RAM					X	4.2.3.3
3.2.1.3.3.2 Read memory protect RAM					X	4.2.3.3
3.2.1.3.3.3 Memory protect enable					X	4.2.3.3
3.2.1.3.3.4 Initialization					X	4.2.3.3
3.2.1.4 Service conditions-electrical					X	4.2.3.3
3.2.1.4.1 Steady-state and transient operation					X	4.2.3.3
3.2.1.4.2 Power require-ments					X	4.2.3.3
3.2.1.4.2.1 Signal interface					X	
3.2.1.4.2.1.1 Input loading					X	

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.4.2.1.2 Output drive					X	
3.2.1.4.3 Warm-up				X		4.2.4
3.2.1.4.4 Thermal design				X		4.2.4
3.2.1.4.5 Processor test				X		4.1.4(3)
3.2.1.4.6 Built-in functions		X				4.1.4(2)
3.2.2 Physical characteristics		X				4.2.2
3.2.2.1 Weight					X	4.2.2
3.2.2.2 Size		X				4.1.4(1)
3.2.2.2.1 Area				X		4.1.4(3)
3.2.2.2.2 Height				X		4.1.4(3)
3.2.3 Reliability			X		X	4.1.4(2)
3.2.4 Maintainability	X					
3.2.4.1 Microprocessor test ability			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.4.2 Testability verification			X			4.1.4(2)
3.2.4.3 Testability models			X			4.1.4(2)
3.2.4.4 Maintenance diagnostic			X			4.1.4(2)
3.2.5 Environmental conditions			X			4.1.4(2)
3.2.5.1 Temperature and altitude environments			X			4.1.4(2)
3.2.5.1.1 Surrounding air temperature			X			4.1.4(2)
3.2.5.1.2 Surrounding air pressure			X			4.1.4(2)
3.2.5.2 Explosive atmosphere			X			4.1.4(2)
3.2.5.3 Humidity and moisture		X				4.1.4(1)
3.2.5.4 Salt-sea atmosphere			X			4.1.4(2)
3.2.5.5 Fungus			X			4.1.4(2)
3.2.5.6 Sand and dust			X			4.1.4(2)

TABLE XV

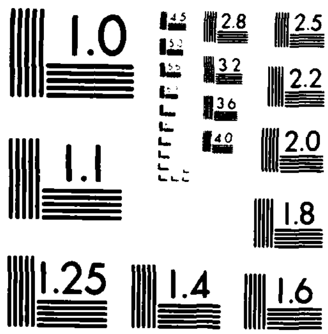
VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.5.7 Vibration environment			X			4.1.4(2)
3.2.5.8 Shock			X			4.1.4(2)
3.2.6 Transportability			X			4.1.4(2)
3.3 Design and construction			X			4.1.4(2)
3.3.1 Materials, processes, and parts			X			4.1.4(2)
3.3.1.1 Design layout				X		4.1.4(3)
3.3.1.2 Microcircuits		X				4.1.4(1)
3.3.1.3 Semiconductors		X				4.1.4(1)
3.3.1.4 Passive devices		X				4.1.4(1)
3.3.1.5 Standard parts		X				4.1.4(1)
3.3.1.6 Nonstandard parts		X				4.1.4(1)
3.3.2 Electromagnetic interface and compatibility				X		4.1.4(3)
3.3.3 Product marking		X				4.1.4(1)
3.3.4 Workmanship		X				4.1.4(1)

AD-A150 584 PROCEEDINGS OF THE TECHNICAL FORUM (3RD) ON THE F-16 MIL-STD-1750A MICROP..(U) AERONAUTICAL SYSTEMS DIV WRIGHT-PATTERSON AFB OH J L PESLER ET AL. 06 MAY 82 2/6

UNCLASSIFIED ASD-TR-82-5011-VOL-2 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963 A

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.3.5 Interchangeability			X			4.1.4(2)
3.3.6 Safety criteria			X			4.1.4(2)
3.3.6.1 Toxicity			X			4.1.4(2)
3.3.7 Human engineering			X			4.1.4(2)
3.3.8 Switching transients			X			4.1.4(2)
3.3.9 Overload protection			X			4.1.4(2)
3.3.10 Thermal analysis			X			4.1.4(2)

TABLE XV

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Analysis 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.4 Documentation		X				4.1.4(1)
3.5 Logistics			X			4.1.4(2)
3.5.1 Maintenance			X			4.1.4(2)
3.5.2 Supply			X			4.1.4(2)
3.6 Precedence	X					

4.1.3 Qualification by similarity. Formal qualification shall be accomplished by using test data from previously developed and qualified items when possible. When qualification by similarity is proposed, the test data from the earlier qualification shall be submitted with design data to substantiate that:

a. The equipment is to perform a similar function in the new application as it did in its earlier qualification.

b. The environmental and operating limits shall be no more demanding or degrading than in the earlier operation.

c. The new item does not incorporate difference that would invalidate the criteria of a or b above.

d. The equipment operated satisfactorily in its earlier airplane application as indicated by its MTBF field failure data.

4.1.4 Method of verification. Verification shall be accomplished by inspection, analysis, demonstration or test, or a combination thereof as defined below:

(1) Inspection. Inspection is defined as a visual verification that the CI as manufactured conforms to the documentation to which it was designed.

(2) Analysis. Analysis is defined as verification that a specification requirement has been met by technical evaluation of equations, charts, reduced data and/or representative data.

(3) Demonstration. Demonstration is defined as an uninstrumented test where success is determined by observation alone. Included in this category are tests that require simple quantitative measurements such as dimensions, time to perform tasks, etc.

(4) Tests. Test is defined as verification that a specification requirements is met by a thorough exercising of the applicalbe element under appropriate conditions in accordance with approved test procedures.

4.1.5 Test conditions.

4.1.5.1 Standard conditions. The following conditions shall be used to establish normal functional performance characteristics:

a.	Ambient temperature	Room ambient (27 +/-10°C)
b.	Surrounding air pressure	Prevailing lab conditions
c.	Humidity	Room ambient up to 90% relative

4.1.5.2 Tolerances for test conditions. The maximum allowable tolerances for test conditions (excluding instrument inaccuracies) shall be in accordance with MIL-STD-810 with the following additions and exceptions:

a. Temperature 2°C

b. Pressure 5% of nominal valur or 0.06 in. Hg, whichever result in the lesser tolerance

c. Relative Humidity 5%
0%

4.1.5.3 Accuracy of test apparatus. The requirements of MIL-STD-810 are applicable except that the maximum allowable error in instrumentation for measurement of the following parameters shall be:

a.	Temperature	2°F
b.	Pressure	Surrounding air, 0.05 in. Hg. Abs; other, 2% of indicated value

All measurements shall be made with instruments of laboratory precision type whose accuracy has been certified. Calibration shall be traceable to the National Bureau of Standards. Unless otherwise indicated, calibration of all measuring and test equipment which control the accuracy of inspection equipment and facilities shall have an accuracy of at least one tenth of the tolerance of the variable to be measured unless such accuracy is not attainable with existing measuring devices. Deviation from the accuracy shall be subject to General Dynamics approval in writing.

4.2 Quality conformance.

4.2.1 Classification of tests.

Acceptance test	4.2.2
Qualification	4.2.3
Examination of product	4.2.3.2
Functional test	4.2.3.3
Reliability	4.2.4
Health and safety assurance	4.2.5

4.2.2 Acceptance tests. The following tests are designated as acceptance tests. Each component submitted to General Dynamics for acceptance shall have satisfactorily passed the following tests.

Examination of product	4.2.3.2
Functional test	4.2.3.3
Reliability test	4.2.4

4.2.3 Qualification tests.

4.2.3.1 Test conditions. Unless otherwise specified, all qualification tests shall be conducted under the standard conditions of 4.1.5.

4.2.3.2 Examination of product. The equipment shall be examined to determine conformance with the applicable drawing and all requirements of this specification for which there are no specific tests.

4.2.3.3 Functional tests. Each equipment shall be operated for a period of time sufficient to establish stability of performance. The performance/functional characteristics of the equipment shall then be measured in accordance with the approved acceptance test specification; and the data recorded. Any equipment which is found out of tolerance shall be rejected.

4.2.3.4 Rejection and retest. Equipment which has failed to meet the acceptance tests of this specification shall be rejected. Final acceptance of equipment on hand or later produced shall not be made until it is determined that items meet all specification requirements. The equipment may be reworked or have parts replaced to correct the cause of

rejection. Full particulars concerning the rejection and action taken to correct the fault shall be submitted with the equipment. Equipment rejected after retest shall not be resubmitted without specific General Dynamics approval.

4.2.4 Reliability.

4.2.4.1 Reliability testing. Each deliverable chip set shall be tested as a set per the temperature cycling requirements of MIL-STD-781, test level F, as modified by paragraph 4.2.4.2 below. The test shall be conducted for at least 50 hours, with the last 25 hours failure free. Each failure occurring during the test shall be analyzed, repaired, documented and reported to General Dynamics. Pattern failures shall require that immediate analysis be performed and corrective action taken on all subsequent chip sets.

4.2.4.2 Temperature cycling. Temperature cycling during all reliability testing shall be in accordance with Figure 1 of MIL-STD-781 utilizing test level F with the following exceptions:

a. During the cooling portion of each cycle -54°C chamber air shall be applied until the point of maximum thermal inertia of equipment under test reaches -40°C as determined by the thermal survey. Chamber air at -40°C shall then be applied and maintained until the temperature of temperature critical components reaches -40°C as determined by thermal survey.

b. Equipment power and maximum temperature chamber air shall then be applied to initiate the heating portion of the cycle.

c. Test time following stablization at high temperature shall be 1 hour and 45 minutes.

d. Vibration requirements of MIL-STD-781 test level F shall not be requirements for Reliability testing.

4.2.5 Health and safety assurance. Design features promoting personnel health and safety will be verified as described in the following paragraphs.

4.2.5.1 Toxicity. Materials of the F-16 weapon system which are listed in Tables G-1, G-2, and G-3 of OSHA Standard 1910.93 will be identified. Designs containing such materials will be analyzed to assure that personnel exposure to the material during adverse operational, maintenance and training conditions is limited to the ceiling values or eight-hour time weighted averages given in the above tables. Tests will be conducted to determine material concentrations wherever analyses are inconclusive.

4.2.5.2 High voltage. Designs with voltages in excess of 30 volts will be analyzed to assure that personnel are protected from inadvertent exposure to such voltages.

4.2.5.3 Hazard protection. Personnel protection from hazards will be verified by a subsystem hazard analysis.

5. PREPARATION FOR DELIVERY

Unless otherwise specified in the purchase order, preparation for delivery of equipment for immediate use at General Dynamics shall be in accordance with the following.

5.1 Preservation, packaging, packing and marking. The equipment furnished to this specification shall be preserved, packaged, packed and marked in accordance with 16PP224.

5.2 Intermediate packaging. The quantity of unit packages to be included in each intermediate package shall be at the option of the manufacturer and as governed by the limitations of the container to be used.

5.3 Packing. Shipments that require overpacking for acceptance by the carrier shall be packed in exterior type shipping containers in a manner that will enure safe transportation at the lowest cost of delivery. Containers shall meet the requirements of the freight classification rules and regulations of the common carriers as applicable to the mode of transportation.

5.4 Marking.. All containers shall be marked to indicate the purchase order number, part number, bill of lading number, case number and total number of pieces. Any special markings as required by carrier rules and regulations shall be applied.

5.5 Special handling, loading techniques and devices. In the event special handling, loading techniques and devices are required to protect the item during moving, storage, installation or removal, there shall be a suitable decal or nameplace affixed to the item delineating the necessary precautions.

6. NOTES

6.1 Address Spaces. The instruction set shall use 16-bit addresses for referencing instructions and data in three logical memory spaces. Each memory space consists of 65,536 words. The three logical memory spaces are the Start Up Read Only Memory (ROM), Main Memory, and Input/Output Registers as shown in Figure 14. The characteristics of each memory space are described below:

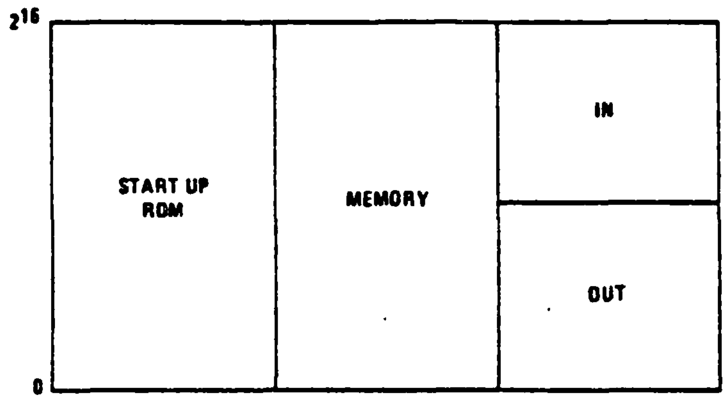


Figure 14 Address Spaces

6.1.1 Start up ROM (optional). The start up read only memory (ROM) address range shall be contiguous starting from address 0 up to a maximum of 65,536, as required by the system application. When the start up ROM is enabled, if an I/O or CPU store function is executed whose address is within the start up ROM, then the store is attempted into main memory. When the start up ROM is enabled, if a read function (instruction or operand) is executed from either I/O or the CPU whose address is to the start up ROM, then the read shall be from the start up ROM. When disabled, the start up ROM cannot be accessed.

The start up ROM will be controlled by logic external to the processor chip.

6.1.1.1 Start-up ROM enable. The CPU shall provide one XIO command to signal that all subsequent read operations should be referenced to the start-up read only memory. A second XIO command signals a return to normal memory operation.

6.1.1.1.1 Enable start-up ROM (4004H). This XIO command may be used to enable the start-up ROM from external logic if it has been implemented.

6.1.1.1.2 Disable start-up ROM (4005H). This XIO command resets the external logic which enables the start-up ROM.

6.1.2 Trigger go counter (optional). A trigger go counter may be implemented by the host system to detect software time-out faults. The CPU has an internally decoded XIO instruction (trigger go reset, reference paragraphs 3.1.2.1.4 and 3.2.1.2.1.5) which may be used by the software to reset the trigger go counter.

6.1.3 Microprocessor chip set interconnections. The microprocessor chip set consists of the CPU, the memory mamagement unit (MMU), and the block protect ram (BPR). These components may be interconnected in at least four (4) configurations. Four possible configurations are described in the following paragraphs.

6.1.3.1 CPU. The CPU chip may be used above without either the MMU or BPR. This example is shown on Figure 3.

6.1.3.2 CPU and BPR. These two components of the chip set are illustrated in Figure 15.

6.1.3.3 CPU and MMU. The CPU may be combined with the MMU for extended memory applications. This configuration is illustrated in Figure 16.

6.1.3.4 CPU, MMU, and BPR. The CPU may be combined with the MMU for extended mamory and the BPR for 1K memory protect resulution. This configuration is illustrated in Figure 17.

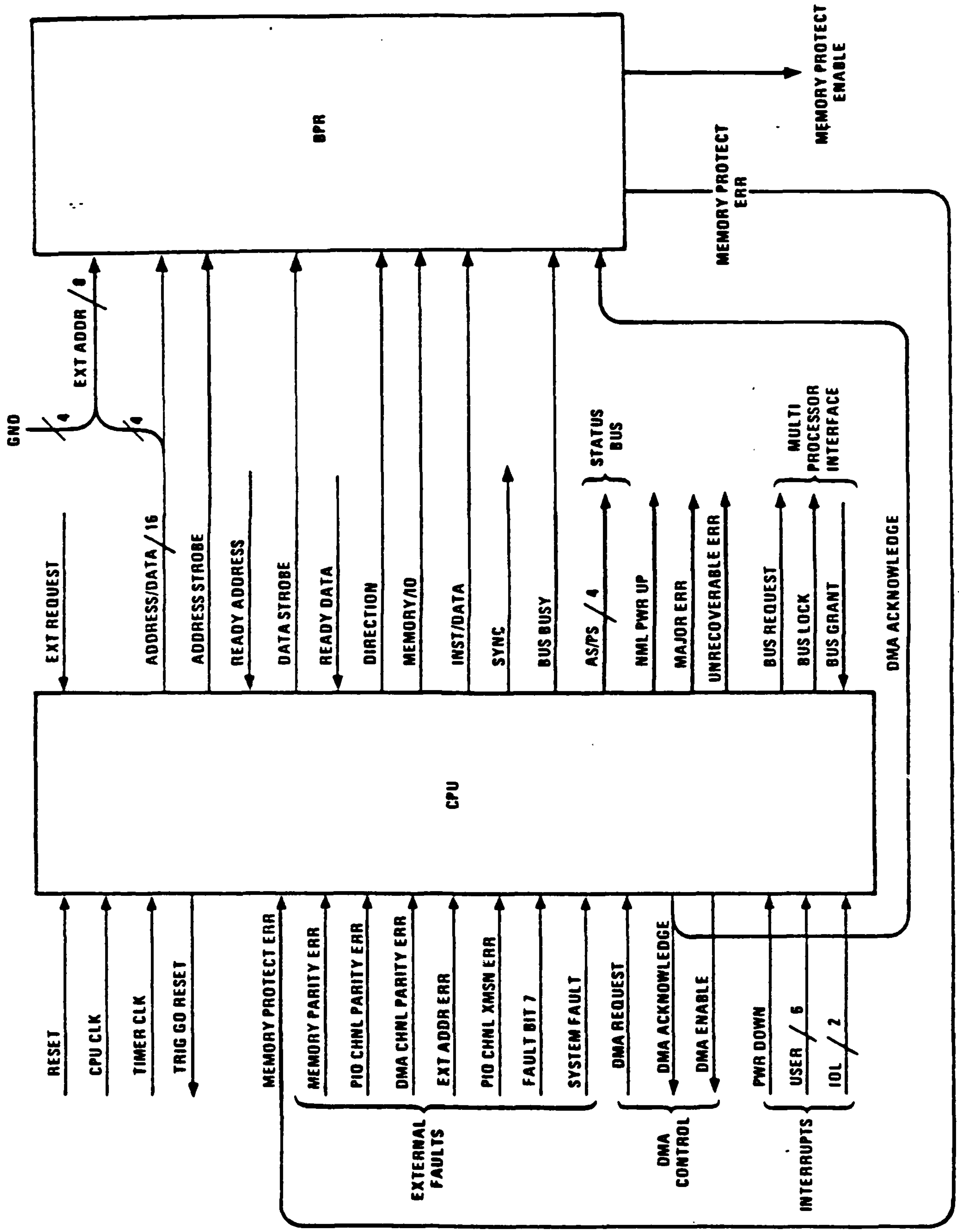


Figure 15 CPU With Optional Block Protect RAM

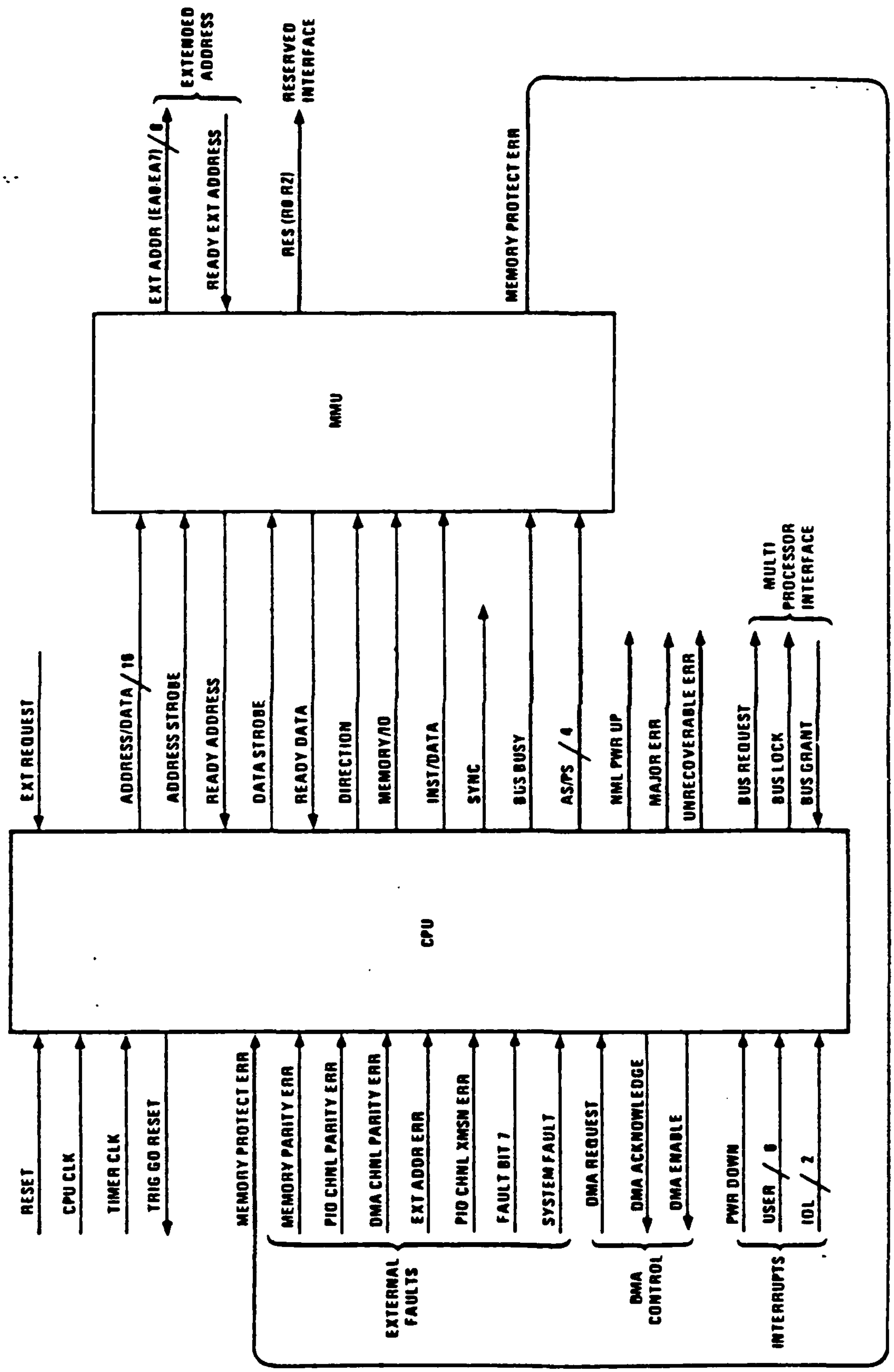


Figure 16 CPU With Optional MMU

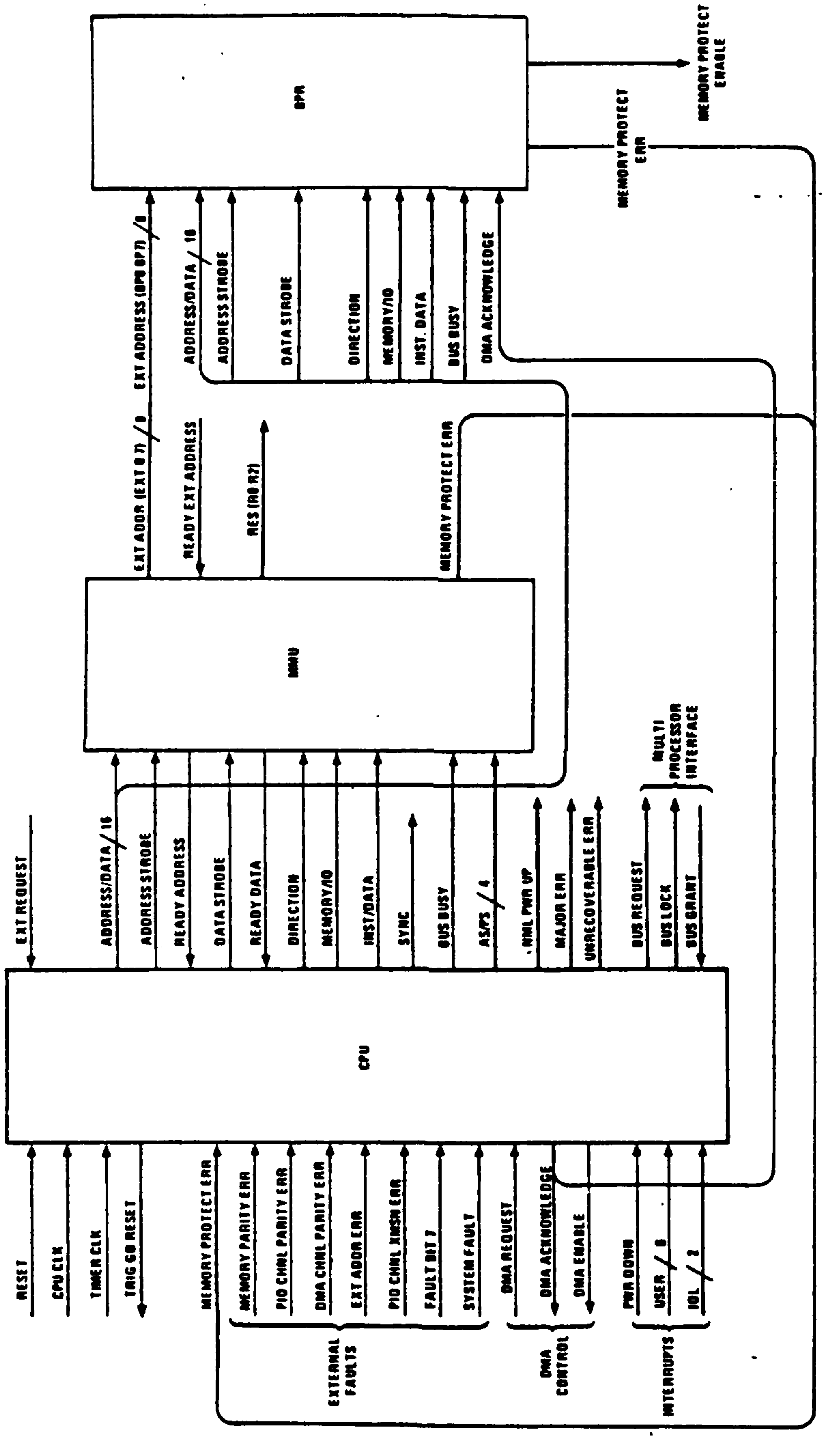


Figure 17 CPU With Optional MMU and Block Protect RAM

6.1.4 Main memory. The instruction set shall use 16-bit logical addresses to provide for referencing of 65,536 words. For a CPU only, physical addresses shall equal logical addresses. The attempted referencing of an undecoded memory address will cause bit 8 of the CPU fault register to set, generate a machine error interrupt, and abort to completion.

6.1.5 Input/Output. The I/O command space shall be divided into 128 channels. Up to 512 commands within each channel group (256 input and 256 output) may be used with each I/O interface. Table XVI lists the 128 I/O channel groups. The attempted execution of an undecoded I/O command shall cause bit 5 of the fault register to be set, generate a machine error interrupt, and abort to completion.

I/O		Channel						Group							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The fields in the input/output address space shall be defined as follows:

I/O:
The bit shall be used to differentiate between input and output register transfers (input = 1).

Channel:
This field shall define which of the 128 I/O channels is being addresed.

Group:
This field shall define the register address within the group.

TABLE XVI INPUT/OUTPUT CHANNEL GROUPS

Output	Usage	Usage
00XX 03XX	80XX 83XX	PIO /
04XX 1FXX	84XX 9FXX	Spare /
20XX	AOXX	Processor & Auxiliary Register Control
21XX 2FXX	AOXX AFXX	Reserved /
30XX 3FXX	BOXX FBXX	Spare /
50XX	COXX	Processor & Auxiliary Register Control
41XX 4FXX	C1XX CFXX	Reserved /
50XX	DOXX	Block Protect Ram
51XX	D1XX	Instruction Page Register Commands
52XX	D2XX	Operand Page Register Commands
53XX 7FXX	D3XX FFXX	Spare /

6.1.5.1 Input. The input instructions transfer data from an external I/O device or an internal special register to a CPU general register. These commands are used to read data from peripheral devices, timers, status word, fault register, discretes, interrupt mask, etc. A full description of the input instructions is given in the MIL-STD-1750 instruction repertoire.

6.1.5.2 Output. The output instructions transfer data from a CPU general register to an external I/O device or special register. These commands are used to write data to peripheral devices, discretes, start and stop timers, enable and disable interrupts and DMA, set and clear interrupt requests, masks and pending interrupt bits, etc. A full description of the output instructions is given in the MIL-STD-1750 instruction repertoire.

6.1.5.3 Dedicated I/O memory locations. If dedicated memory locations are used to communicate information to and/or from an I/O channel, these locations shall be consecutive memory locations starting at an implementation defined location. Locations (40H) through (4FH) are otional for I/O usage.

6.2 Address/data fault detection. The requirements for the detection of address/data faults divided between the microprocessor and the host equipment. The CPU shall implement fault detection for the following:

6.2.1 Unimplemented addresses. All memory and I/O addresses within the microprocessor and throughout the user subsystem shall be exhaustively decoded for the purposes of detecting unimplemented addresses. The Host equipment shall be responsible for decoding all addresses which are implemented outside of the microprocessor. The microprocessor will determine that an addressing fault has occurred if and only if the address error discrete is activated and the address is not implemented in the microprocessor.

6.2.2 Memory protect errors. The CPU will monitor a discrete memory protect signal from the memory subsystem in order to determine that a memory protect error has occurred. The CPU will monitor all data transactions including DMA. The determination that a location is write protected will be the responsibility of the memory management function of a microprocessor with MMU paragraph 3.2.1.2.2 or of the block

protect RAM described in paragraph 3.2.1.2.3. If both memory protect options are implemented simultaneously within the same subsystem, the memory protect signal shall be "wire-or'd" between the functions so that a memory protect violation will be detected if at least one function activates the memory protect line.

6.2.3 Parity error detection. Parity generation/checking shall be an option of the user equipment. The CPU will monitor the parity error discrete signal from the memory subsystem in order to determine that a memory parity error has occurred. The CPU will monitor all data transactions including DMA and XIO instructions.

6.3 User implemented options. The CPU will interface with the following user implemented options in accordance with MIL-STD-1750.

6.3.1 Direct memory access. DMA may be implemented as required for a particular application. The CPU will be capable of disabling direct memory access (DMA) to memory through the clearing of DMA enable which is controlled by the following XIO instructions:

(1) DMA enable, Reference paragraph 3.2.1.2.1.6.3 (2) DMA disable. The DMA Request line is activated by a user device wanting service. Service is granted via the DMA acknowledge. DMA Request is held high until DMA Acknowledge is present. When DMA acknowledge is present, the user may take possession of the bus. After the user bus transaction has been completed, the user releases DMA Request. After the CPU recognizes DMA Request is not present, DMA Acknowledge is removed. After another user may pull DMA Request low to request a bus transaction.

6.3.2 Input/output interrupt code register (IOIC). The input/output interrupt code registers, if implemented, are used to indicate which channel generated the input/output interrupt. One register is assigned for each of the two input/output interrupts. Each register is set by hardware to reflect the address of the highest priority channel requesting that level of interrupt. The address shall be 00H for the channel number 0, 0FH for channel number 15, 7FH for channel number 127, etc. The IOICs shall not be altered once the interrupt sequence has commenced until they are read by an I/O instruction.

SPARE								CHANNEL CODE							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The following commands shall be implemented if the IOIC option is used:

6.3.2.1 Read input/output interrupt code, level 1 (0A000H). This command inputs the contents of the level 1 IOIC register into register RA. The channel number is right justified.

6.3.2.2 Read input/output interrupt code, level 2 (0A001H). This command inputs the contents of the level 2 IOIC register into register RA. The channel number is right justified.

6.3.3 Console input/output. Console input/output, if required by the user application, shall be implemented through use of the following XIO commands:

6.3.3.1 Clear console (4001H). This command clears the console interface.

6.3.3.2 Console output (4000H). The 16-bit contents (2 bytes) of register RA are output to the console. The eight most significant bits (byte) are sent first. If no console is present, then this command will be treated as a NOP.

6.3.3.3 Console input (C000H). This command inputs the 16-bits (2 bytes) from the cnsole into reguster RA. The eight most significant bits of RA shall represent the first byte.

6.3.3.4 Read console status (C001H). This command inputs the console interface status into register RA. The status is right justified.

6.3.4 Memory fault status register (MFSR). Memory fault status register provides the page register selection designators associated with memory faults. The page register designators (below) captured by the MFSR are valid for the memory reference causing the fault.

LPA				RESERVED							IO	AS			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

LPA:
Address of page register within the set

Reserved:
Bits 4 through 10 shall always be zero.

IO:
Instruction/Operand page register set selector (1 = instruction).

AS:
Address of selected group.

The MMU shall respond to the following XIO command for the MFSE:

6.3.4.1 Read memory fault register (0A00DH). This command transfers the 16-bit contents of the memory fault status register to RA. The fields within the memory fault status register shall delineate memory related fault types and shall provide the page register designators associated with the designated fault.

16PP379A
7 JULY 1981

AVIONIC PROCESSOR STANDARD INSTRUCTION SET ARCHITECTURE REQUIREMENTS

A S 2 82 0336

16PP379A
7 JULY 1981

AVIONIC PROCESSOR STANDARD INSTRUCTION

SET ARCHITECTURE REQUIREMENTS

APPROVED BY:

P. L. CURRIER
ENGINEERING CHIEF
COMPUTERS & DISPLAYS

W. C. BOOTON
ENGINEERING MANAGER
SENSORS, PROCESSING & DISPLAY

J. D. ENGELLAND
ENGINEERING MANAGER
SYSTEMS ENGINEERING

1.0 SCOPE

The purpose of this document is to standardize the implementation of the MIL-STD-1750 instruction set architecture in equipment delivered to or built by General Dynamics/Fort Worth Division.

2.0 APPLICABLE DOCUMENTS

2.1 Government documents. The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered as superseding requirements.

SPECIFICATIONS

MIL-STD-1750A 2 July 1980	Sixteen-bit computer instruction set architecture

3.0 REQUIREMENTS

All implementations of MIL-STD-1750 processors utilized by General Dynamics shall include the items as delineated in Section 3.1. Additional requirements, as outlined in the applicable specification, shall be implemented per Section 3.2.

3.1 Compulsory requirements. All mandatory requirements of MIL-STD-1750 shall be implemented as detailed. In addition, this section delineates the options and utilization of spares for MIL-STD-1750 processors. Any changes in the operation or function of these items shall be subject to prior approval by General Dynamics.

3.1.1 Fault register. In addition to the requirements as outlined in MIL-STD-1750, spare bit 7 shall be set to one (1) when the Software Fault Indicator times out. This Software Fault Indicator shall be as defined by the applicable processor specification.

3.1.2 Interval timers. Both Timer A and Timer B shall be implemented per MIL-STD-1750.

3.1.3 Memory parity. Parity associated with the memory system shall be odd. A detection of even parity within this memory system shall cause bit 2 of the fault register to be set.

3.1.4 Interrupt control. A minimum of sixteen levels of interrupt shall be implemented as detailed in MIL-STD-1750 paragraph 4.6 and Table VIII.

3.1.4.1 Mandatory interrupts. Of the interrupts listed in Table VIII, the following eight (8) interrupts are defined fully by MIL-STD-1750 and this document.

INTERRUPT NO.	NOMENCLATURE
0	Power Down (cannot be masked or disabled)
1	Machine Error (cannot be disabled)
3	Floating Point Overflow
4	Fixed Point Overflow
5	Executive Call (cannot be masked or disabled)
6	Floating Point Underflow
7	Timer A
9	Timer B

NOTE: Interrupts 0 and 5 cannot be Masked or Disabled. Loading the interrupt mask register has no effect on these two interrupts.

3.1.4.2 Input/output level interrupts. Interrupts number 12 and 14, if used, must conform to the usage described in MIL-STD-1750.

3.1.4.3 Spare interrupt. Of the six remaining interrupts described in Table VIII of MIL-STD-1750, the interrupt allocations outlined in the following paragraphs shall be observed.

3.1.4.3.1 Reserved interrupts. The following interrupts shall be reserved for usage by the hierarchical MIL-STD-1553 bus network.

INTERRUPT NO.	NOMENCLATURE
8	Avionics Mux
10	Display Mux
11	Weapons Mux

3.1.4.3.2 User interrupts. Interrupts numbered 2, 13 and 15 are unallocated spares which may be further defined by the subsystem specification.

3.1.5 Execute input/output (XIO). The following provides clarification to various XIO commands:

A. The execution of the XIO command 4003 MPEN (Memory Protect Enable) shall disable global memory protection.

B. The XIO instructions defining Programmed Input/Output shall be optional and only implemented if specified in the applicable specification.

C. If an illegal XIO command is encountered as part of a VIO chain, the following action shall occur:

(1) The illegal I/O command bit of the fault register (FT_5) is set to a one.

(2) The VIO chain is terminated, and the illegal XIO is treated as a NOP.

The VIO chain is terminated upon attempted execution of an illegal XIO command, and the illegal I/O command bit (FT_5) is set. This termination shall not affect execution of preceding XIO commands which are part of the VIO chain being executed.

D. When executing the XIO command 2005 SPI (Set Pending Interrupt) register, the 16-bit contents of RA shall be OR'ed with the pending interrupt register.

3.1.5.1 Mandatory XIO commands. Mandatory I/O commands shall be implemented per MIL-STD-1750.

3.1.5.2 Optional XIO commands. The following optional XIO commands shall be implemented in all processors delivered to General Dynamics per MIL-STD-1750.

4006	DMAE/DMA Enable
4007	DMAD/DMA Disable
4008	TAS/Timer A Start
4009	TAH/Timer A Halt
400A	OTA/Output Timer A
400C	TBS/Timer B Start
400D	TBH/Timer B Halt
400E	OTB/Output Timer B
C00A	ITA/Input Timer A
C00E	ITB/Input Timer B
30XX	Reserved for Avionics MUX outputs
B0XX	Reserved for Avionics MUX inputs
31XX	Reserved for Display MUX outputs
B1XX	Reserved for Display MUX inputs
32XX	Reserved for Weapons MUX outputs
B2XX	Reserved for Weapons MUX inputs

The remaining optional I/O commands shall be implemented if required as delineated by MIL-STD-1750.

3.1.5.3 "User defined XIO functions". The user defined XIO functions shall be implemented within the framework as defined by MIL-STD-1750 if required by the processor specification.

3.1.6 Double precision integer divide. When executing a double precision integer divide instruction, a fixed point overflow shall occur if the divisor "DO" is zero, or if the dividend is $8000\ 0000_{16}$ and the divisor is $FFFF\ FFFF_{16}$.

3.1.7 Negate register/absolute value. When executing single or double precision negate register or absolute value instructions, the contents of the destination register(s) shall always be updated independent of the overflow condition.

3.1.8 Branch to executive. When executing the Branch To Executive instruction, bit 5 of the pending interrupt register shall not be set.

3.1.9 Single precision integer divide. When executing a single precision integer divide with a 32-bit dividend, a fixed point overflow shall occur if the divisor equals zero, or if a positive quotient exceeds $7FFF_{16}$ or if a negative quotient is less than 8000_{16}.

3.1.10 Memory indirect indexed single precision instructions. When executing a memory indirect indexed single precision instruction, the derived address shall be [A + (RX)].

3.1.11 Instruction counter relative instructions. When executing an instruction counter relative instruction, the single precision derived address shall be [D + (IC-1)].

3.1.12 Single and double precision shifts. When executing single and double precision shift instructions, if the absolute value of the shift count is greater than 16 or 32 bits, respectively, a fixed point overflow shall occur and the instruction shall be treated as a NOP.

3.1.13 Floating point addition and subtraction. All floating point results shall be truncated toward negative infinity. Floating point add and subtract arithmetic shall be done as though all registers have infinite length with the results truncated toward negative infinity, such that [A + B = A - (-B)].

3.1.14 Floating point divide. When executing any floating point divide instruction, a zero floating point number divided by any nonzero floating point derived operand shall produce zero without any floating point overflow event. Zero divided by zero shall cause a floating point overflow.

3.1.15 Built-in-functions. The 8-bit primary op code $4F_{16}$ shall be reserved for implementing Built-In-Functions. Processor response to the complete 16-bit op code $4FXX_{16}$ shall be to execute the Built-In-Function indicated by the 8-bit op code extension XX_{16}, if that function is physically implemented; otherwise, the processor shall set bit 9 (Illegal Instruction) of the Fault Register and generate a Machine Error interrupt.

3.2 Additional options. Additional options as required by the applicable subsystem specifications shall be implemented per MIL-STD-1750.

3.2.1 Memory block protect. If required by the processor specification, memory block protect shall be implemented per MIL-STD-1750. The following XIO commands shall be used:

4003	MPEN/Memory Protect Enable
50XX	LMP/Load Memory Protect
D0XX	RMP/Read Memory Protect

The reset (power-up) state of the processor shall establish all of main memory as globally protected. This global protection shall remain in effect until the execution of the Memory Protect Enable XIO instruction. All memory addresses in the memory protect mechanization shall represent physical memory locations.

3.2.2 Software fault indicator. This indicator shall be implemented as defined by the applicable processor specification. Bit 7 of the fault register shall be set when this indicator times out.

4. QUALITY ASSURANCE PROVISIONS

Quality Assurance shall be as defined in the applicable subsystem development specification.

DRAFT

Dɛ... -
~~SPECIFICATION~~ NO. 16PP456
CODE IDENT: 81755
Date: 10 January 1982

INTERIM PROCESSOR DESIGN REQUIREMENTS

DRAFT

PREVIOUS PAGE IS BLANK

A 9 D 62 0336

16PP456
10 January 1982

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SCOPE

1.1 <u>Scope</u>. This specification establishes the performance, manufacture, and test requirements for embedded computers using the Z8002 microprocessor.

1.2 <u>Purpose</u>. The purpose of this specification is to standardize the hardware and software requirements necessary to insure a successful phased approach to incorporate MIL-STD-1750 microprocessors developed in accordance with General Dynamics specification 16ZE181.

2.0 APPLICABLE DOCUMENTS

2.1 <u>Government documents</u>. The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered as superseding requirements.

SPECIFICATIONS

<u>Military</u>

MIL-E-5400 31 Oct 1975	Electronic equipment, Airborne General specifications for
MIL-T-18303	Test Procedures; Preproduction and Acceptance for Aircraft Electronic Equipment, Format of
MIL-M-385100	Microcircuit Design, General Specification For

STANDARDS

<u>Military</u>

MIL-STD-883B Notice 4 4 Nov 1980	Test Methods and procedures for Microelectronics
MIL-STD-965 Notice 1 22 December 1978	Parts Control Program

STANDARDS

Military

MIL-STD-1750 Notice 1 July 1981	Sixteen Bit Instruction Set Architecture

2.2 Non-government documents. The following documents of exact issue shown form a parta of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered as superseding requirements.

SPECIFICATIONS

General Dynamics

16PP379A 7 July 1981	Avionic Processor Standard Instruction Set Architecture Requirements
16ZE181 22 Sept 1981	MIL-STD-1750 Microprocessors, General Specification For

3.0 REQUIREMENTS.

3.1 Item definition. For the purpose of this specification, an embedded computer shall be partitioned into a Central Processing Unit (CPU), Fault Register, System Timers, Interrupt Subsystem, Direct Memory Access control circuitry, optional memory block protection mechanism, and a memory I/O subsystem. These functions are equivalent to a MIL-STD-1750 CPU or a MIL-STD-1750 CPU with Block Protect RAM as specified in 16ZE181, and the Memory/IO subsystem specified in the prime item equipment specification.

3.1.1 Central processing unit. The CPU shall be a Z8002 microprocessor or equivalent built to MIL-M-38510/520-02. The CPU will have sixteen (16) general purpose registers, a system stack pointer, a program counter, a flag and control word, and a program area status pointer. The CPU and required elements of the imbedded computer are shown in Figure 1.

3.1.2 Fault register. The fault register shall be used to indicate the origin of machine error interrupts. The logical "OR" of the fault register bits shall be used to generate the machine error interrupt. The contents of the fault register shall be read and cleared in a single instruction. Information required for the setting of bits in the fault register shall be derived from the following error signals: Memory Protect Error, Memory Parity Error, External Address error, Trigger Go Timeout, and System Fault. An example of Fault Register implementation is shown in Section 6.

3.1.3 System timers. Two timers shall be considered as part of the system processing function. The timers shall be implemented using an AMD 9513 System Timing Controller (or equivalent). The two timers will be referred to as Timer A and Timer B. The timers shall be sixteen (16) bits in length and shall generate vectored interrupts to the CPU as described in paragraph 3.2.1.3. An example timer implementation is illustrated in Section 6.

3.1.4 Interrupt subsystem. The interrupts shall be divided into four classes based on the interrupt structure of the CPU and MIL-STD-1750. A cross reference of the interrupts defined for MIL-STD-1750 and the CPU is given as Tables I and II.

3.1.4.1 Power down. The power down interrupt definition shall be in accordance with MIL-STD-1750 Interrupt 0. The

power down interrupt is the highest priority interrupt and cannot be masked or disabled. This interrupt shall be implemented with the CPU non-maskeable interrupt (NMI) if required by the host equipment specification.

3.1.4.2 Machine errors. The machine error interrupt definition shall be used in accordance with MMIL-STD-1750 except as modified herein. The machine error interrupt shall be implemented through the CPU non-vectored interrupt which is disabled through the flas and control word (FCW). The machine error interrupts shall be generated through the fault register described in paragraph 3.2.1.2.

3.1.2.3 User and timer interrupts. The user and timer interrupts shall be implemented using the CPU vectored interrupt as shown in Table II. The vectored interrupt controller shall be implemented using the AMD99519 Interrupt Controller or equivalent built to MIL-M-38510/42x-02.

3.1.4.4 Software traps. The following MIL-STD-1759 interrupts shall be implemented using software traps as defined for the CPU: Executive Call, Illegal Instruction, and Privileged Instruction fault. Software traps cannot be disabled or masked.

3.1.4.5 Program generated interrupts. The software shall be able to activate an interrupt by setting a bit in the interrupt controller Interrupt Request Register.

3.1.5 Direct memory access control. DMA may be implemented as required for aparticular application. The CPU will be capable of disabling direct memory access (DMA) to memory through the setting of a discrete signal (Direct Memory Access Enable) which is controlled by XIO commands. When enabled the DMA Control shall perform the bus acquisition for DMA devices.

3.1.5.1 DMA request. The DMA Control shall accept assynchronous DMA bys requests from DMA devices after it has disabled the CPU address/data bus.

3.1.5.2 DMA acknowledge. If enabled by the software, the DMA control shall acknowledge DMA requests after it has disabled the CPU address/data bus.

Figure 1 System Processing Function

TABLE I

MACHINE ERROR INTERRUPTS

MIL-STD-1750			Z8000	
FAULT	INTERRUPT	FAULT REGISTER	FAULT INTERRUPT	REGISTER
Memory Protect Fault	IPT1	FR 0	NVI*	ZFR 15
DMA Mem Prot Fault	IPT1	FR 1	NVI*	ZFR 14
Memory Parity Error	IPT1	FR 2	NVI*	ZFR 13
Illegal I/O ADD	IPT1	FR 5	NVI*	ZFR 10
Software Fault Indicator	IPT1	FR 7	NVI*	ZFR 8
Illegal Memory Add	IPT1	FR 8	NVI*	ZFR 7
Illegal Instruction	IPT1	FR 9	EI TRAP	------
Priviledged Inst.	IPT1	FR10	PI TRAP	------
System Fault	IPT1	FR15	NVI*	ZFR 0

*NVI is effectively masked by the Flag Control Word (FCW) Bit 11

NVI = Non-vectored Interrupt

EI = Extended Instruction

PI = Priviledged Instruction

TABLE II

INTERRUPT DEFINITIONS

	MIL-STD-1750		Z8000	
FUNCTION	INTERRUPT	MASK REGISTER	INTERRUPT	MASK REGISTER
Power Down	IPT 0	----	NMI	----
Machine Error	See Table I			
User 0	IPT 2	IM 2	VI 0	VIM 0
Executive Call	IPT 5	----	SC TRAP	----
Timer A	IPT 7	IM 7	VI 1	VIM 1
User 1	IPT 8	IM 8	VI 2	VIM 2
Timer B	IPT 9	IM 8	VI 3	VIM 3
User 2	IPT 10	IM 10	VI 4	VIM 4
User 3	IPT 11	IM 11	VI 5	VIM 5
User 4	IPT 13	IM 13	VI 6	VIM 6
User 5	IPT 15	IM 15	VI 7	VIM 7

IM = Imterrupt Mask

VI = Vectored Interrupt

VIM = Vectored Interrupt Mask

SC = System Call

3.1.5.3 DMA enable. The DMA enable idscrete output may be provided to indicate the readiness of the DMA control to respond to DMA requests.

3.1.6 Memory protection mechanism. The memory block protection mechanism shall provide the memory/IO subsystem with a memory protect error signal suitable for write protecting data in 1024 word blocks. The memory block protection mechanism shall consist of a memory element for storing block protection information from the CPU and a logic element as shown in Figure 2.

3.2 Characteristics

3.2.1 Performance. The performance characteristics of the system processing function shall be specified herein.

3.2.1.1 Z8000 throughput. Processor throughput shall be as specified in the host equipment specification.

3.2.1.2 Fault register (ZFR). Hardware faults detected in the host equipment shall be reported through the fault register as specified in 16ZE181 and in accordance with MIL-STD-1750 and this specification. The Fault Register is read and cleared by the following X-10 command:

Memory Protect		Mem Par	0	0	I/O Add	0	TG	Mem Add	0	0	0	0	Bit	0	System Fault
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Sys 1 Sys 2

Bit 15: CPU Memory Protect Error

Bit 14: DMA Memory Protect Error

Bit 13: Memory Parity Error

Bit 12: Always 0

Bit 11: Always 0

Bit 10: Illegal XIO Address Error

Bit 9: Always 0

Bit 8: Trigger Go Timeout

gg

Figure 2

Memory Protection Mechanism

BLOCK

PROTECT

RAM

LOGIC

ELEMENT

Bit 7: Memory Address Error

Bit 6,5,4, and 3: Always 0

Bit 2: Built In Test

Bit 1: Always 0

Bit 0: System Fault

3.2.1.2.1 Memory protect error. This discrete line from the Block Protect RAM shall be used to set ZFR(15) if the CPU is in control of the external bus when activated by the host equipment. Otherwise ZFR(14) shall be set when the discrete is activated.

3.2.1.2.2 Memory parity error. This discrete line shall be used to set ZFR(13) when activated.

3.2.1.2.3 Address error. This ~~discrete~~ ZFR line from the host equipment shall be used to set ZFR(~~7~~) for an illegal memory address error. This discrete line shall be used to set ZFR(1) for an illegal XIO address error.

3.2.1.2.4 Trigger go timeout. This discrete line shall be used to set ZFR(8) when activated by the host equipment.

3.2.1.2.5 System fault. This discrete line shall be used to set ZFR(0) when activated by the host equipment. and or fault reg bit 1

3.2.1.3 Timers. Under software control of the CPU, the timers shall be started, stopped, loaded and read. Provisions shall be made for suspending and resuming the counting sequence through a discrete control line which shall be made available in the host equipment test connector.

3.2.1.3.1 Timing scale. Timer A shall increment every ten (10) microseconds while enabled. Timer B shall increment every hundred (100) microseconds while enabled.

3.2.1.3.2 Timing range. The timers shall use a binary counting sequence beginning at count 0 and progressing to count 65,535. An interrupt request shall be made whenever a timer increments from a count of 65,535 to a count of 0.

3.2.1.3.3 Timer initialization. The AM9513 timer may be initialized as follows to approximate Timer 1 and Timer B. Timer A is defined as Counter 1. Timer B is defined as Counter 2. The input frequency at X2 is expected to be 1.0 Mesahertz. The Prescaler is set to count BCD to divide the 1.0 Mesahertz down to 100 Kilohertz (10 microseconds) at F2 for Timer A to 10 Kilohertz (100 microseconds) at F3 for Timer B. Both timers should be initialized to start of 0000H and increment to FFFFH then interrupt. The 1750A timers do not decrement. A suggested detailed initialization is illustrated in Section 6. 6

3.2.1.4 Interrupt subsystem. The four classes of interrupts shall be implemented as follows:

3.2.1.4.1 Power down interrupt. The power down interrupt will be provided by the host subsystem if required by the prime item equipment specification.

3.2.1.4.2 Machine error interrupt. The machine error interrupt shall occur in response to the setting of a bit in the ZFR as defined in 3.2.1.2 above.

3.2.1.4.3 User and timer interrupts. The user and timer interrupts shall be serviced through the 9519 vectored interrupt controller as defined herein.

3.2.1.4.3.1 Interrupt request register (IRR). The IRR shall store pending interrupt requests. A bit in the IRR is set whenever the interuppt request is received at that interrupt level. Bits in the IRR shall also be settable under software control from the CPU, thus permitting software generated interrupts. The bits in the IRR are resettable under software control. An IRR bit will automatically be cleared when its interrupt is acknowledged.

3.2.1.4.3.2 Interrupt service register (ISR). The ISR contains one bit for each interrupt request. It will be used to indicate that a pending interrupt has been acknowledged and to mask all lower priority interrupts. The ISR shall provide for either automatic clearing of the ISR bit during the interrupt acknowledge cycle or for manual clearing under software control.

3.2.1.4.3.3 Interrupt mask register (IMR). The IMR will be used to enable or disable the individual vectored interrupt requets. Bits in the IMR correspond with interrupt requests

and shall be loaded, set and cleared under software control. A mask bit that is set will not disable the IRR, and an interrupt request which arrives while a corresponding mask bit is set will cause an interrupt later when the corresponding mask bit is cleared.

3.2.1.4.4 Software traps. The following software traps may be used to provide equivalent functions to those specified in MIL-STD-1750.

3.2.1.4.4.1 Executive call. The Executive Call interrupt (Interrupt 5) defined in MIL-STD-1750 shall be implemented via the CPU System Call (SC) software trap feature of the Z8002 ISA.

3.2.1.4.4.2 Illegal instruction. The Illegal Instruction interuppt, which is one of the machine error interrupts defined in MIL-STD-1750, shall be implemented using the Extended Instruction (EI) trap feature of the Z8002 ISA.

3.2.1.4.4.3 Priviledged instruction (PI). The Privileges Instruction interrupt, which is another of the machine error interrupts defined in MIL-STD-1750, shall be implemented using the Privilege instruction trap feature of the Z8002 ISA.

3.2.1.4.5 Program generated interrupts. The software may generate a vectored interrupt to signal execution complete of a software function. The software interrupt is generated by setting a bit in the AM9519 interrupt controller Interrupt Request Register (IRR). This allows the software to generate any of the six (6) user interrupts for use by software.

3.2.1.5 Direct memory access. ~~The direct memory access logic shall convert the DMA control signals to Z8002 Bus Request and Bus Acknowledge signals.~~ The DMA Control shall perform the bus acquisition after the current bus transaction has been completed for direct memory access devices. Bus acquisition is performed in response to the DMA request line if the DMA is enabled.

6

3.2.1.5.1 DMA request. The DMA control shall accept asynchronous bus requests from DMA devices via the DMA Request signal. ~~The DMA request signal shall be active high when a DMA device is requesting the bus.~~ If enabled the DMA Acknowledge shall respond to a DMA request at the end of the current address/data bus transaction or within TBD CPU clock cycles.

Q 3.2.1.5.2 DMA acknowledge. The DMA acknowledge signal shall be set high when the CPU has disabled its address/data bus in response to a DMA request. DMA acknowledge shall remain high ~~until~~ DMA request has been removed.

3.2.1.5.3 DMA enable. DMA shall be disabled at CPU power up or reset. Thereafter the response of the CPU to DMA request will be controlled by execution of the DMA enable/disable command (B605H). The DMA enable output discrete shall be active high when the DMA is enabled.

add 3.2.1.6 Block protect RAM (BPR). If required by the user application, the memory block protect mechanism shall protect up to 64K words of memory using a RAM with 64 bits dedicated to protecting CPU memory and 64 bits dedicated to protecting DMA memory as illustrated in Figure 3a. An example implementation is illustrated in Section 6.

3.2.1.6.1 Block protect status. The Block Protect logic element shal use the following XIO address to load and read the CPU and DAM Block Protect Status. Memory protect selectively disables writing into memory depending upon whether the bit protecting that 1K portion of memory is one or zero. One in the bit location disables writing and Zero enables writing. An Enable Latch is used to enable the contents of the interal RAM to protect the designated areas against writing data into them. If this enable is not set the entire CPU and DMA memory is globaly protected.

(a) Load Block Protect RAM (B70XH). --Load Block Protect RAM loads the contents of the accumulator eight (8) least significant bits into the addressed BLock Protect RAM byte.

(b) Read Block Protect RAM (B70XH). --Read Block Protect RAM transfers the contents of the addressed location in BPR to the least significant eight (8) bits of the accumulator.

3.2.1.6.2 Global memory protection. The memory protection ~~mechanism shall be enabled by Power~~ on Reset and Block Protect RAM Disable XIO Instruction. e

At Power on or Reset ~~Initially~~ all memory shall be write protected regardless of the status of the Block Protect RAM bits until the protection mechanism is enabled.

Figure 3

BLOCK PROTECT RAM ORGANIZATION

XIO Address	Bit Number	Memory Protection Range
Z002 address	7,6,5,4,3,2,0	
6E1E		E000-FFFF
6E1C		C000
6E1A		A000
6E18		8000
6E16		6000
6E14		4000
6E12		2000
6E10		0000-1FFF
6E0E		E000-FFFF
6E0C		C000
6E0A		A000
6E08		8000
6E06		6000
6E04		4000
6E02		2000
6E00		0000-1FFF

make 1 CPU

other DMA

Most Significant Block

Least Significant Block

check the original and insert

more table 7 to

4.0 QUALITY. Not applicable

5.0 PREPARATION FOR DELIVERY. Not applicable.

6.0 NOTES

6.1 Address spaces. The Z8002 instruction set uses 16 bit addresses for referencing 65,535 bytes of information. When the 15 most significant bits of Z8002 address is combined with the status information provided on each data transfer cycle, and with the operational state of the CPU, it is possible to distinquish among six 32,768 word memory address spaces and two 32,768 word input/output address spaces as shown below:

	SYSTEM MODE	NORMAL MODE
INSTRUCTIONS	32K	32K
DATA	32K	32K
STACK	32K	32K
I/O	32K	
SPECIAL I/O	32K	

Z8002 Address Spaces

6.1.1 Memory. The Z8002 distinquishes between memory references for innstructions, data, and stack in either system or normal mode. These logical memory address spaces should be mapped into a single 65,536 word physical address space in accordance with the following table:

Logical Address Space	Physical Address Space
Instructions	0000H-7FFFH
Stack,Data	8000H-FFFFH

6.1.2 Input/Output. The attempted execution of an undecoded XIO command shall cause bit 10 of the fault register to be set, generate a machine error interrupt, and abort to completion.

6.1.2.1 Input. The input instructions transfer data from an external XIO device or an internal special register to a CPU general register. These commands are used to read data from peripheral devices, timers, status word, fault register, discretes, interrupt mask, etc.

6.1.2.2 Output. The output instructions transfer data from a CPU general register to an external XIO device or special register. These commands are used to write data to peripheral devices, discretes, start and stop timers, enable and disable interrupts and DMA, set and clear interrupt requests, masks and pending interrupt bits, etc.

6.1.2.3 Input/Output channel bit assignments. Table III illustrates the relationship between Bus address bits and the Z8002 processor address lines for XIO and Memory.

6.1.2.3.1 Memory and XIO transactions. Memory and XIO transfers shall be word transactions, (R/W=0) for all memory and XIO addresses. (B/WR = 0)

TABLE III

Z8002 Input/Output Channel Groups

1750A ADD/DATA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bus Address bits are related in the Z8002 address bits as shown below:																
Bus Address lines:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Z8002 Addr lines:	*	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Module Channel Select — Register Select

Note: Z8002 Data lines correspond in the Bus Address lines.

* For all XIO transactions and operations 1750A Address Bit Zero (0) is a one (1) for input and a Zero (0) for output. In

the Z8002 implementation the equivalent bit is always a one (1). In this case the Z8002 Read/Write line may be used to differentiate between input and output instructions.

state code data partition

6.1.2.3.2 Memory address. Memory transactions shall derive the most significant bit of address (BA 15) from ST2/ and ST3/, (Z8002 Status bit 2 and 3) if required. During XIO transactions B15 will be 1. Direction and Instruction/Data may be used in decoding XIO commands where ncessary. Since the ISB of the Z80C address (ZAD0) is not included it is a "don't care".

6.1.2.3.3 Module channel select address. Bus address bits (10-8) are suggested for module or channel select address.

6.1.2.3.4 Register select address. Bus address bit (3-0) are suggested for register select address.

6.1.2.3.5 XIO command map. The following map, shown in Table : is suggested for I/O commands and data transfers.

TABLE IV

Table of Suggested XIO Codes for the Interim Processor System Processing Function

Bus Address	Z8002 Address	
B000H	6000H	PIO
B40XH	680XH	
B600H	6C00H	Timer Data
B601H	6C02H	Timer Command
B602H	6C04H	Interrupt Controller Data
B603H	6C06H	Interrupt Controller Command
B604H	6C08H	Read and Clear Fault Register
B605H	6C0AH	**DMA Enable/Disable
B606H	6C0CH	**Memory Protect Enable/Disable
B607H	6C0~~F~~EH	Intercomputer Interrupt

e put in AMX etc.

** The Enable/Disable function may be implemented by using the least significant data bit. 0=Disable, 1=Enable.

remove in its entirety

6.1.2.3.6 Block protect RAM. The following map, shown in Ta
V, VI, VII is suggested for the Block Protect RAM for 64K of
RAM, and 64K of DMA RAM. If the memory protect RAM is implem
using two 16X4 bit. RAMs placed side by side for 8K of protec
per RAM word used XIO address pertaining to this function (Re
or Write memory protect RAM) applies to 8K words of protected
memory. The lower 8 bits of each XIO word accesses the memor
protect RAM, the upper 8 bits are unused.

use

TABLE V

CPU Memory Protect Address Map

RAM Byte	BUS IO Address	Z8002 IO Address	Memory Protect Block
0	B700H	6F00H	0000H-1FFFH
1	B701H	6F02H	2000H-3FFFH
2	B702H	6F04H	4000H-5FFFH
3	B703H	6F06H	6000H-7FFFH
4	B704H	6F08H	8000H-9FFFH
5	B705H	6F0AH	A000H-BFFFH
6	B706H	6F0CH	C000H-DFFFH
7	B707H	6F0FH	F000H-FFFFH

TABLE VI

DMA Memory Protect Address Map

8	B708H	6F10H	0000H-1FFFH
9	B709H	6F12H	2000H-3FFFH
10	B70AH	6F14H	4000H-5FFFH
11	B70BH	6F16H	6000H-7FFFH
12	B70CH	6F18H	8000H-9FFFH
13	B70DH	6F1AH	A000H-BFFFH
14	B70FH	6F1CH	C000H-DFFFH
15	B70FH	6F1FH	F000H-FFFFH

In summary the relation of each RAM byte to its designated memory blcok is as follows where bit 7 is the MSB and bit 0 the ISB.

TABLE VII

Block Protect RAM Map
(Protect RAM 8 Bits Wide)

Bit	Applicable Memory Block	
7	1st	1K words
6	2nd	1K words
5	3rd	1K words
4	4th	1K words
3	5th	1K words
2	6th	1K words
1	7th	1K words
0	8th	1K words

A given 1K word memory block is protected from write accesse when its applicable RAM bit is set in a one.

Unimplemented address shall be reported for

change

6.1.3 Unimplemented addresses. All memory and XIO addresse within the CPU and throughout the user subsystem shall be ex haustively decoded for the purposes of detecting unimplement or erroneous addresses. ~~The CPU will monitor~~ all ~~data~~ trans actions including DMA. Figure 4 illustrates a possible dec for the interim processor.

bus

6.2 User implemented options. The PSU will interface with the following user implemented options in accordance with MIL-STD-1750.

6.2.1 Parity error detection. Parity generation/checking shall be an option of the user equipment. The CPU will mor the Parity error discrete signal from the memory subsystem order to determine that a memory parity error has occurred.

6.3 Example implementations. Some examples are included t illustrate the operation of various elements of the 1750A Interim Processor. Examples are included to assist system hardware and software designers in implementing the interi processor. These are examples only and should be thoroug analyzed to determine the accuracy and fit to the specific application.

6.3.1 Timer initialization and operation. The first item to be accomplished is to set up the AM9513 System Timing Controller after a power up or reset.

6.3.1.1 Timer initialization. The following will be an illustration of abbreviated marchine language code for setting up the AM9513 as Timer A and Timer B in the 1750A Interim Processor configuration.

	HEX CODE	Description
1.	3FXX	Out'Rd.(0).Rs
	6C02	Timer Command, in Rd
	FFFFH	Data, Reset Code for Am9513, in Rs
2.		Out
	6C02H	Timer Command
	FFFFH	Data, Set Master Mode Register in 16 bit bus
3.		Out
	6C02H	Timer Command
	F100H	Set Master Mode per Table VIII
4.		Out
	6C02H	Timer Command
	FF01H	Load Data Pointer Reg to counter 1 mode register
5.		Out
	6C02H	Timer Data
	000,0,1100,0,0,1,0,1,101	Binary Output mode in counter 1 mode register per Table XIX
	0C20H	hex equivalent of binary above
6.		Out
	6C02H	Timer Command
	FF02	Load Data Pointer Reg to counter 2 mode register
7.		Out
	6C00H	Timer Data
	000,0,1100,0,0,1,0,1,101	Binary Output mode in counter 2 mode register per Table XIX plus change 1
	0D2DH	hex equivalent of binary

8.		Out
	6C02H	Timer Command
	FF09H	Set Load Data Pointer Reg to counter 1 load register
9.		Out
	6C00	Timer Data
	0000H	Load, counter 1 load register = 0
10.		Out
	6C02H	Timer Command
	FF0A	Set load data pointer register to counter 2 load register
11.		Out
	6C00H	Timer Data
	0000H	Load, counter 2 load register = 0

TABLE VIII

System Timing Controller Master Mode Register Setup

Master Mode Bit Reference 1750A	Z8002	Description	Bit value (1/0)
0	15	Scaler Control = BCD (decimal)	1
1	14	Data Pointer Control=Disable Increment	1
2	13	Data Bus Width = Sixteen Bits	1
3	12	FOUT Gate = Frequency Out on Chip	0
4	11	FOUT Divider = 1	0
5	10		0
6	9		0
7	8		1
8	7	FOUT Source = F1	0
9	6		0
10	5		0
11	4		0
12	3	Compare 2 Enable=Disabled	0
13	2	Compare 1 Enable=Disabled	0
14	1	Time of Day = Disabled	0
15	0		0

TABLE IX

System Timing Controller Counter Mode Register Setup

Counter Mode Bit 1750A	Counter Mode Bit Z8002	Description	(1/0 Bit value)
0	15	Gating control = No Gating	0
1	14		0
2	13		0
3	12	Count Source Selection = count on rising edge	0
4	11	Clock=F2	1
5	10		1
6	9		0
7	8		0
8	7	Count Control = Disable Special gate	0
9	6	Count Control = Reload from load	0
10	5	" " = Count Repectitively	1
11	4	' " = Binary Count	0
12	3	Output Control = Active low Terminal count pulse	1
14	1		0
15	0		1

Table XIX Change 1
Change Count Source Selection Clock=F3

6.3.1.2 Timer operation. The following will be an example of abbreviated machine language code for operation of the AM9513 as Timer A and Timer B in the 1750A Interim Processor configuration.

AFTER THE TIMERS HAVE BEEN INITIALIZED THEY CAN BE STARTED

1.		Out
	6C02H	Timer Command
	FF63H	Load and Arm counter 1 and 2

TO LOAD AND ARM COUNTER 1 (TIMER A) WITH A COUNT OTHER THAN ZERO

1.		Out
	6C02H	Timer Command
	FF09H	Set load data pointer register in counter 1 load register
2.		Out
	6C00H	Timer Data
	XXXXH	Starting count data for counter 1 (Timer A)
3.		Out
	6C02H	Timer Command
	FF61H	Load and Arm counter 1

TO LOAD AND ARM COUNTER 2 (TIMER B) WITH A COUNT OTHER THAN ZERO

1.		Out
	6C02H	Timer Command
	FF0AH	Set load data pointer register to counter 2 (Timer B)
2.		Out
	6C00H	Timer Data
	XXXXH	Starting count data for counter 2 (Timer B)
3.		Out
	6C02H	Timer Command
	XXXXH	Load and arm counter 2

TO STOP COUNTER 1 (TIMER A)

1.		Out
	6C02H	Timer Command
	FFC1H	Disarm counter 1

TO RESTART COUNTER 1 (TIMER A) FROM WHERE IT HAD BEEN STOPPED.

1.		Out
	6C02H	Timer Command
	FF21H	Arm counter 1

TO STOP COUNTER 2 (TIMER B)

1.		Out
	6C02H	Timer Command
	FFC2H	Disarm counter 2

TO RESTART COUNTER 2 (TIMER B) FROM WHERE IT HAD BEEN STOPPED

1.		Out
	6C02H	Timer Command
	FF22H	Arm counter 2

TO READ COUNTER 1 (TIMER A) INTO REGISTER IN THE Z8002

1.		Out
	6C02H	Timer Command
	FF11H	Load data pointer register to counter 1 hold register
2.		Out
	6C02H	Timer Command
	DESTH	Input data in Z8002 register

TO READ COUNTER 2 (TIMER B) INTO REGISTER IN THE Z8002

1.		Out
	6C02H	Timer Command
	FF12H	Load data pointer register to counter 2 hold register
2.		Out
	6C02H	Timer Command
	FFA2H	Save counter 2
3.		In
	6C00H	Timer Data
	DESTH	Input data to Z8002 register

6.3.2 Interrupt controller initialization. Suggested initialization of the 9519A.

The first item to be accomplished is to set up the Internal Z8002 parameters.

The Flag and Control Word (FCW) myst be set to disable all interrupts except the non maskable interrupt (NMI). (See load control ease 6-70.6-76 in the Brown CPU Technical Manual). Disabling the Interrupts allows the program to initialize the AM9519A with the appropriate parameters and not be subject to extraenous Interrupts.

Next load the Program Status Area Pointer. The Program Status Area must start on a 256 Byte (128 word) boundary since the interrupt levels are assigned fixed memory locations beginning at a 256 byte boundary in the Z8002 Microprocessor.

The AM9519A is setup as follows:

MODE	Mode Register AM9519A SETUP	BIT	VALUE	Z8002 BIT	1750A BIT
Priority Mode	Fixed	0	0	0	15
Vector Selection	Individual	1	0	1	14
Interrupt Mode	Interrupt	2	0	2	13
GINT Polarity	Active low	3	0	3	12
IRFQ Polarity	Active low	4	0	4	11
Register	Auto Clear	5	1	5	10
Preselection	Auto Clear	6	1	6	09
Master Mode	Mask all	7	0	7	08

INITIALIZE THE AM9519A AT POWER UP

```
HEX
CODE

1.             Out'R.(D 0).Rs
      6C06H    Int.Cont.Command
      0000H    Data Reset Conde for AM9519A in Rs
        --
      --  !
       !  -----Active code data, command for the AM9519A
       !
       !
       --------Zero by convention. The eight (8) most
               significant bits are not used on the AM9519A.


2.             Out
      6C06H    Int.Cont.Command
      00AFH    Data. Load Mode Register bits 5.6 and Clear
               bit 7.
               Set Register Pointer to AUTO CLEAR and DISABLE
               ALL INTERRUPTS to allow undisturbed setup.

3.             Out
      6C04H    Int.Cont.Data
      00FFH    Data. Load Auto Clear Register to clear all bits
               in the Interrupt Service Register automatically
               after receipt of the Interrupt Acknowledge from
               the Z8002 hardware.

               LOAD MODE REGISTER TO CHARACTERIZE INTERRUPTS

4.             Out
      6C06H    Int.Cont.Command (Load Mode Register bits M4.M0)
      0080H    Data. Set Priority Mode = Fixed, Individual
               Vector.  Interrupt Mode GINT Polarity = Active
               low.
               IRFO Polarity = Active low
```

LOAD RESPONSE MEMORY WITH VECTORED INTERRUPT LEVEL 0-7

5.		Out
	6C06H	Int.Cont.Command
	00F0H	Data. Load Response Memory Location for level 00H-00H. One Byte Identifier
6.		Out
	6C06H	Int.Cont.Command
	00F1H	Data. Load Response Memory Location for level 01H=01H One Byte Identifier
7.		Out
	6C06H	Int.Cont.Command
	00F2H	Data. Load Response Memory Location for level 02H=02H One Byte Identifier
8.		Out
	6C06H	Int.Cont.Command
	00F3H	Data. Load Response Memory Location for level 03H-03H. One Byte Identifier
9.		Out
	6C06H	Int.Cont.Command
	00F4H	Data. Load Response Memory Location for level 04H=04H One Byte Identifier
10.		Out
	6C06H	Int.Cont.Command
	00F5H	Data. Load Response Memory Location for level 05H=05H One Byte Identifier
11.		Out
	6C06H	Int.Cont.Command
	00F6H	Data. Load Response Memory Location for level 06H=06H One Byte Identifier

12. Out
6C06H Int.Cont.Command
00F7H Data. Load Response Memory Location for level 07H=07H
One Byte Identifier

13. Out
6C06H Int.Cont.Command
0020H Data. Clear all IMR bits

12. Out
6C06H Int.Cont.Command
00ADH Data. Set Auto Clear and Set Mode bit 7 (Enable Interrupts in the AM9519A).

Load the FCW Register in the Z8002 to enable all interrupts to test the Interrupt system and verify Interrupt Operation. use software to set the IRR for each interrupt bit. the processor should recognize the interrupt and go to the respective location in the Z8002 Program Status Area.

Example Logic Diagrams

6.4 Replacement processor characteristics. In order to incorporate a MIL-STD-1750 microprocessor in place of the Z8002 and its associated circuitry, the host equipment should make the provisions described below.

6.4.1 Signal interface. The MIL-STD-1750 microprocessor will interface with conventional low-power Schottky TT! signal levels as defined herein. Note positive signal excursions are specified with reference to the primary power source (VCC).

6.4.1.1 Input loading. MIL-STD-1750 signal current supplied from an input mode will be not greater than 800 microamperes when the input voltage is within the range of -0.6 to +0.8 volts. The mode will sink not more than 40 microamperes whenever the input voltrage is in the range of 2.0 to VCC+0.5 volts.

6.4.1.2 Output drive. MIL-STD-1750 microprocessor signal current available at an output mode will not be less than 400 microamperes at an output boltage of 2.4 volts. The output mode will be capable of sinking 2000 microamperes while maintaining an output boltage of less than 0.5 volts.

6.4.2 Service conditions - electrical. Electrical service conditions shall be in accordance with MIL-R-5400 except as modified herein.

6.4.2.1 Steady-state and transient operation. The MIL-STD-1750 microprocessor will provide the specified performance when the power inputs are within the steady state limits of Table VII. The MIL-STD-1750 microprocessor will not be damaged for any power excursion within the transient limits of Table VII.

TABLE X Steady-State and Transient Voltage Limits

	Steady-State limit	Transient Voltage limit
primary power	+5volts+-10%	0 to + 7 volts

6.4.2.2 Power requirements. Total power requirements will be within the limits specified for each class and type of processor as called out in 16ZE181.

SPECIFICATION NO. 16ZE165A
PART I OF TWO PARTS
CODE IDENT NO. 81755
DATE: 8 APRIL 1981

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR THE F-16 INTEGRATED JOVIAL J73 SUPPORT SOFTWARE SYSTEM

This document contains Technical Data considered to be a resource under ASPR 1-319.1(b) and DoD Directive 5400.7 and is not a "record" required to be released under the Freedom of Information Act.

APPROVED:

D. E. SUNDSTROM
FIRE CONTROL SOFTWARE
ELEMENT MANAGER

MSIP II Avionics
L. CRITTENDEN, JR.

PREVIOUS PAGE IS BLANK

1. SCOPE

1.1 Identification. This specification establishes the requirements for performance, design, test, and qualification of a series of computer programs identified as the F-16 Integrated Jovial J73 Support Software System (referred to in this specification as the Support Software System). This Support Software System is used to develop Operational Flight Programs for various computerized F-16 on-board avionic subsystems using the programming language J73. The Support Software System is an important and integral part of the design and maintenance tools supporting advanced F-16 weapons systems.

1.2 Functional summary. The F-16 Integrated J73 Support Software System shall provide all the computer programs required to support the programming of selected target computers in the Higher Order Programming Language (HOL) J73. This system shall consist of a Jovial J73 compiler interfaced with one or more code generators. It shall also include the assembler(s) and linkage editors(s) capable of supporting several different target machine instruction set architectures. The Support Software System shall also provide all the compilers, data bases and other processors and tools necessary to provide for its own enhancement and maintenance on the host computer system. The terms compiler, assembler and linkage editor used throughout this specification shall be extended as follows. References to the term compiler shall be considered to apply to each combination of compiler target computer independent processor and compiler code generation processor developed under this specification. Likewise, each use of the terms assemble and/or linkage editor shall be considered to apply to each assembler and/or linkage editor developed under this specification.

2. APPLICABLE DOCUMENTS

2.1 Government documents. The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be

considered as superseding requirements. Documents referenced within the documents cited herein shall not be applicable to this specification because of such reference. Such sub-referenced documents will be used as design recommendations. Compliance with the design recommendations of the handbooks and approval of the configuration will be documented at the design reviews.

SPECIFICATIONS

Military

MIL-STD-1589B 06 June 1980	Jovial (J73)

2.2 Non-Government documents. The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the specifications referenced herein and the contents of this specification, the General Dynamics specifications listed shall supersede the contents of this specification.

None.

3. REQUIREMENTS

This section contains the performance and design requirements for the F-16 Integrated J73 Support Software System. The programs developed under this specification shall provide all the functions necessary to develop and maintain Operational Flight Programs in the Programming Language J73.

3.1 Computer program definition. The F-16 Integrated Support Software System shall provide the capability to compile, assemble and link a number of J73 program modules and compool data files into a single target computer load file. This load file shall be capable of being read by General Dynamics laboratory computer support equipment for subsequent loading and execution in various target computers. The support Software System shall also provide the capability for General Dynamics or its customer to maintain, modify and enhance any part of the Support Software System. The Support Software System shall be programmed in Jovial J73 (as defined by MIL-STD-1589) or alternately, in another HOL if approved by General Dynamics. The HOL used to program the Support Software System shall be referred to in this specification as the Implementation Language. Minimum use of languages other than the Implementation language (such as assembler language) will be allowed for performing I/O and other host environment interactions. Use of such languages shall not exceed five percent of the Support Software System. All compilers, preprocessors, data files, pattern generators and other programs and data used in developing or testing any part of the Support Software System shall be considered part of the Support Software System.

3.1.1 Interface requirements. The Support Software System shall be capable of being hosted on an IBM 370, 3033 or 4300 series computer operating under either the OS/MVS or VM/VS1 operating systems. The design should be such that no source modifications are required to operate the system under either of the two indicated operating environments. If this is not possible, the contractor shall localize such operating system

dependencies to a minimum number of modules and shall deliver and test modules compatible with each system.

3.1.1.1 Interface block diagram. For reference, a representative functional interface diagram showing the external relationships of the Support Software System to the overall software development process is provided in Figure I-1. The relationships among the various internal functions of the Support Software System are shown in Figure I-2. Figure I-1 and I-2 are representative only and are not intended to be definitive interface diagrams.

3.1.1.2 Detailed interface definition. A description of the interface between the Support Software System and its program development environment, and a description of the interface among the various components of the Support Software System shall be as specified in Appendices A through G. Additional definition of this interface is provided in the following subparagraph.

3.1.1.2.1 Host invocation of programs. Each separate program of the Support Software System shall be placed into execution by the host operating system or alternatively, by a calling program. Each program shall close all opened files and shall return all storage and buffers dynamically acquired from the operating system before returning to its caller. Each program shall be designed such that once a module has been placed in memory, that module may be invoked an unlimited number of times. No execution of a load module in memory shall be affected by the results of its previous execution. Each program shall use a return code of zero to indicate no program errors were detected and that only information type messages were produced. Each program shall be placed into execution by the operating system as a stand alone job step, or by another program through the use of the IBM Macro instructions ATTACH or LINK and CALL. When invoked by the above mentioned methods, each program shall adhere to standard IBM linkage conventions. Each program shall accept an input parameter string from the IBM Job Control Language (JCL) stream or

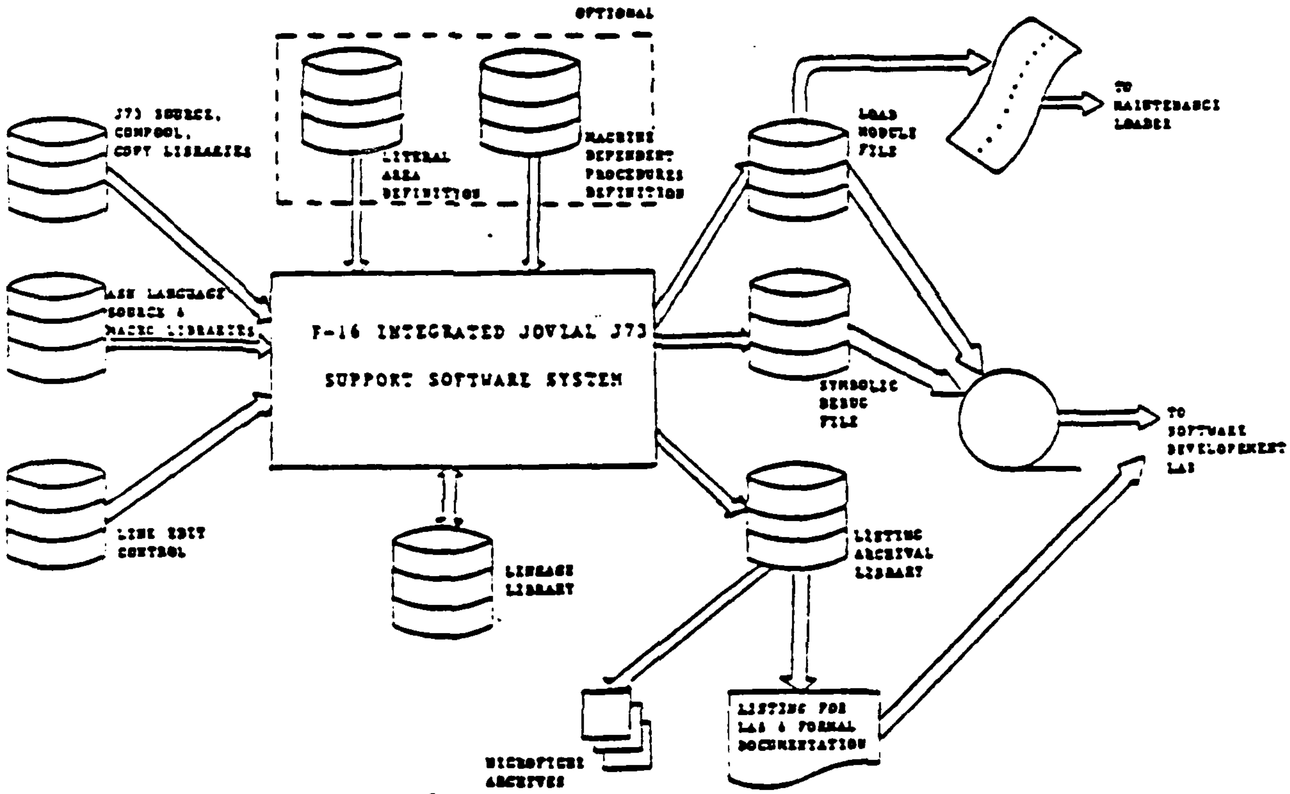


Figure I-1 External Interface Diagram

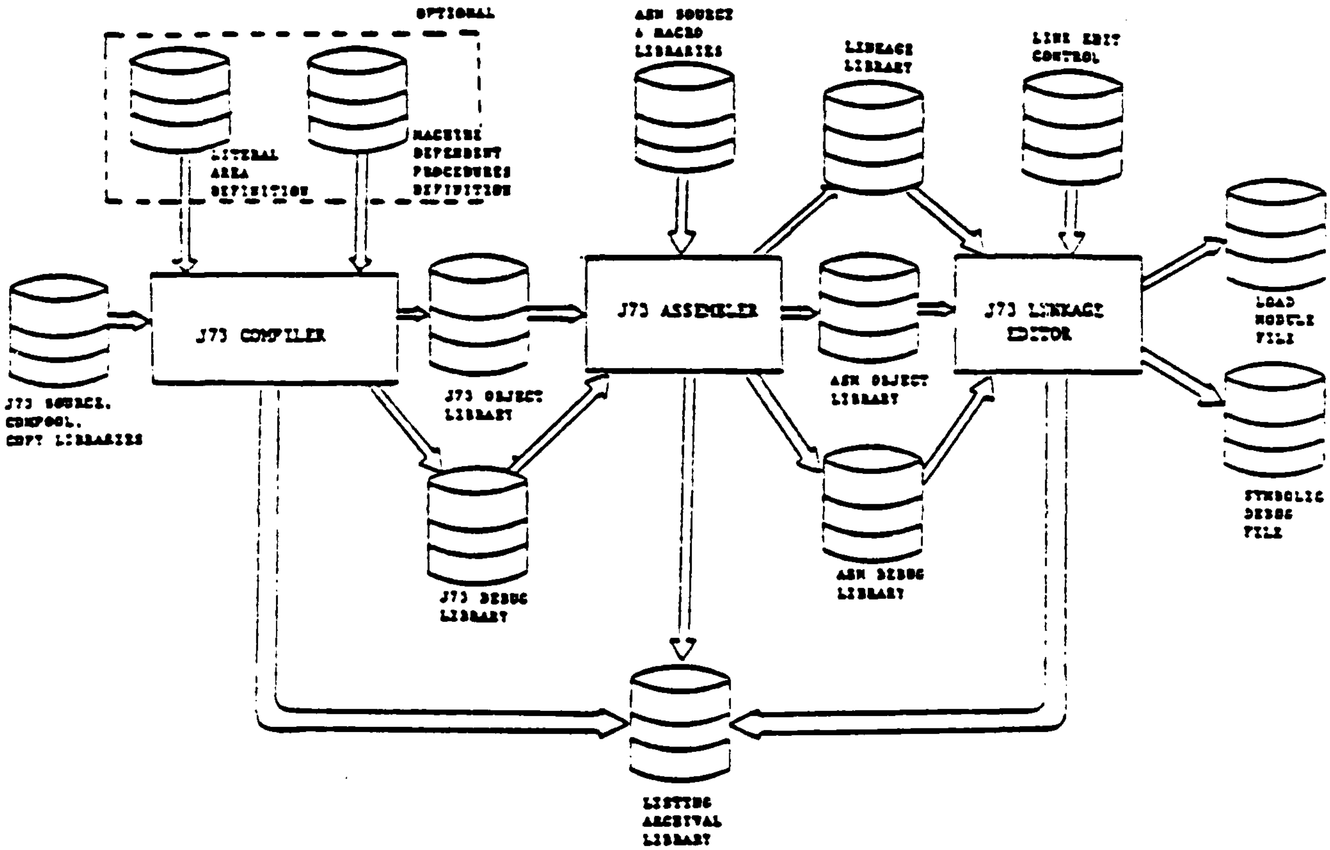


Figure I-2 Internal Functional Interface Diagram

invoking program and shall output a return code indicating the highest severity of any errors detected. Alternate main programs shall be delivered for the compiler and assembler which shall be capable of being called as Implementation Language procedures. These programs shall accept as input a character string parameter and shall produce an integer output parameter representing the program return code.

3.1.1.2.2 Input/Output Interface. The Support Software System shall use standard IBM OS sequential and partitioned data sets for the input and output files of each program. Input and output files (including printable files) shall consist of fixed-length, blocked records. The block size used for input and output files shall be specified by the JCL invoking each program. Modification to any program shall not be required to change the block size of any input or output file. Limits on the maximum allowed block size for any input or output file should be avoided by use of buffer allocation utilities provided by the operating system. All sequential files used by each program shall be defined in a device-independent manner. When a sequential data set is provided to a Support Software System program, the JCL will specify a sequential data set name, or a partitioned data set name and a member name using parentheses. Partitioned data set will reside on various direct-access devices. When a partitioned data set is provided to a Support Software program, the JCL will specify only the partitioned data set name. The program will use operating system access services to locate the desired member. Although desirable, the support Software System need not support block size changes to the temporary files required for its operation. Files of this type will also reside on various direct access devices. Output files designated for printing shall consist of 133 character records with a standard ASA carriage control character in the first character position. The Support Software System shall use the ASA page eject control character to force printing on a new page. Multiple blank lines shall not be used for this purpose. Each Support Software System program shall accept an input parameter which shall indicate the maximum number of lines to be placed on a page before advancing to the next page.

3.2 Detailed functional requirements. The detailed functional requirements imposed on the Support Software System shall be specified in this section.

3.2.1 J73 compiler. The Jovial J73 compiler shall process programs written in the Jovial J73 language, defined by MIL-STD-1589. The entire J73 language and all directives defined in that standard shall be implemented. The compiler shall be designed as two independent functional components. Input programs shall be processed by a target machine-independent component which shall output a well defined intermediate language representation of the program. This output shall then be processed by a code generation component which shall generate symbolic target machine assembler language code as output. Although designed as two separate components, the compiler shall execute as one single program.

3.2.1.1 Compiler inputs. The compiler shall accept its inputs from external files supplied through the JCL stream. The compiler shall also accept a character string option list consisting of as many as 100 characters as an input parameter supplied through the JCL command stream.

3.2.1.1.1 Source input. The compiler shall be capable of locating its primary source input from one of several alternative sources. If NNNN is the module name supplied in the input parameter string, then the compiler shall locate its input according to the following rules, applied in the precedence specified below.

a. From the JCL DD card //NNNN specifying a sequential file,

b. From member NNNN of the partitioned data set specified by the JCL DD card //J73LIB,

or c. From the JCL DD card //SYSIN specifying a sequential file.

If the compiler cannot find its input in any of the above locations, it shall return to its caller with a severe error return code.

3.2.1.1.2 Compool data file input. The compiler shall be capable of locating any referenced compool data files from two alternative sources. If CCCC is the name of the compool file invoked, then the compiler shall locate that file according to the following rules applied in the precedence specified below.

a. From the JCL DD card //CCCC specifying a sequential file

or b. From member CCCC of the partitioned data set specified by the //CMPLLIB.

Each compool file referenced may be either a symbolic compool or may be a preprocessed compool (described in paragraph 3.2.1.2.8). The compiler shall determine the type of each compool file provided solely through the contents of the file.

3.2.1.1.3 Copy file input. The compiler shall be capable of locating any invoked "copy files" from two alternative sources. If XXXX is the name of copy file invoked, then the compiler shall locate that file according to the following rules applied in the precedence specified below.

a. From the JCL DD card //XXXX specifying a sequential file

or b. From member XXXX of the partitioned data set specified by the JCL DD card //COPYLIB.

3.2.1.2 Compiler processing. The processing performed by the compiler shall include all the functions necessary to convert J73 input programs into symbolic assembler language for a given target machine. The functions performed by the compiler shall be decomposed into independent components communicating through a well-defined intermediate language and associated data structures.

a. Target-independent component - The compiler target-independent component shall process J73 programs in a target independent fashion and shall interface with a distinct code-generation component. The output of this component shall be designed such that multiple code generation components can be integrated with this target-independent component. The target independent component shall be capable of being integrated with code generators for target machines whose word length are either 16, 24 or 32 bits.

b. Code generation components - A compiler code genreation component is required for each target machining instruction set architecture. The compiler code generation component shall generate symbolic assembler language code suitable, after assembly, for execution on the target machine instruction set architecture.

3.2.1.2.1 Run-time support requirements. In order to implement certain J73 language constructs, the assembler language code output from the compiler may generate subroutine calls to support modules which execute at run-time. Such run-time support calls shall be limited to those J73 constructs which, on a specific target machine architecture, would be impractical or inefficient to implement as "in-line" code. All run-time support modules shall be programmed in assembler language. All such modules shall be totally relocatable and shall be capable of supporting re-entrants, recursive J73 programs.

3.2.1.2.2 Literal area. In order to reduce the amount of storage required for various commonly used constants, the code generator shall generate code that accesses a predefined literal area. The definition of this literal area shall be an external input file to the compiler or alternatively, a single compiler source module. This file shall consist of a list of absolute memory locations and the contents of those locations for a target computer. The purpose of this data shall be for frequently referenced program constants as well as numerical and logical patterns typically required by the code generator

but not explicitly found in source programs. The generated code shall access this literal area when appropriate for efficient target machine programs. A predefined literal area, consisting of TBD constants, shall be supplied with the compiler along with a target machine run-time support module for loading the literal area with the proper constant data. User oriented documentation shall be provided in sufficient detail so that programmers may generate literal area definitions tailored to specific applications.

3.2.1.2.3 Machine-dependent subroutines. The compiler shall be designed to accommodate user specification of machine-dependent subroutines and functions. When possible, the code generated by a machine-dependent subroutine shall make use of the target machine state (such as register contents) at the time of its invocation. General Dynamics prefers the definition of machine-dependent subroutines and functions to be obtained by the compiler from an external file provided as an input to the compilation process. Alternately, such subroutine definitions may be built into the compiler provided it is done so in an orderly and modular fashion. No more than 1% of a compiler (measured at the module level) shall require modification for the incorporation of (or modification to) any machine-dependent subroutine or function. Compiler documentation shall provide detailed, user oriented instructions for the incorporation of (or modification to) any machine dependent subroutine or function into the compiler.

3.2.1.2.4 Optimizations for efficient code. The compiler shall optimize the generated output code. The efficiency loss of the assembly language code produced by the compiler shall not exceed fifteen percent in size or speed as compared to quality hand-generated code. The compiler shall implement (but not be limited to) those techniques listed in Table I-I as required. Additional optimizations should be provided as described in the following subparagraphs.

TABLE I-I TYPICAL OPTIMIZATION TECHNIQUES

a. Constant folding and propagation

b. Eliminations of common subexpressions

c. Reordering of arithmetic expressions

d. Statement re-ordering

e. Subscript evaluation

f. Operator strength reduction

g. Loop test replacement

h. Loop fusion

i. Invariant operator processing

j. Elimination of unaccessible code

3.2.1.2.4.1 Register, subroutine and parameter conventions. Register usage, subroutine linkage and parameter passing conventions shall be optimized for programming of real-time airborne avionic applications.

3.2.1.2.4.2 Special statement processing. The compiler shall provide input options or directives which prohibit the use of the abort, exit, or recursive procedure features in any procedure invocation or statement. When processing an input program under this option, the compiler shall denote as an error any statement containing these commands. The compiler shall assume other external procedures do not implement these language features.

3.2.1.2.4.3 Nested-procedure processing. The compiler shall provide input options or directives which prohibit the use of the declaration of nested procedures. When processing an input program under this option, the compiler shall indicate as an error any usage of this language feature. The compiler shall assume other external procedures do not implement this language feature.

3.2.1.2.4.4 Additional subroutine call optimization. The compiler shall be designed to provide any input options or directives necessary to limit the use of those J73 constructs which contribute to the generation of object code supporting linkage to subroutines.

3.2.1.2.5 Symbolic debug support. The compiler shall produce information for an assembler, and linkage editor to supporting symbolic debugging at the target machine level. Generation of this data shall be controlled by an input parameter or option. This information might typically include symbol tables, statement tables, entry/exit tables, etc. The exact content and form of this output is specified in Appendix F.

3.2.1.2.6 Preprocessed Compools. The compiler shall accept an input parameter which shall indicate that normal processing of its input shall be by-passed, and instead, a preprocessed compool shall be generated. The input to the compiler when

invoked with this option shall consist only of a single compool module. The compiler shall process this module and produce a output with condensed encoding of the module that is suitable for input to the compiler performing normal module processing. The compiler shall be designed so that when a compool file is referenced by a J73 module, the compilation time for that module is substantialy reduced if the compool is provided in preprocessed form. When an entire preprocessed compool has been referenced the compiler should only retrieve the necessary information to compile the current input program. The object program output by the compiler shall not be affected by the form in which its referenced compools are provided. When operating in the preprocess mode, the compiler need not produce an object program that allocates any storage defined by the input compool. This output shall always be generated when a compool is input in the nonpreprocess mode.

3.2.1.2.7 Error recogition and recovery. The compiler shall correctly process and produce target assembler code implementing any syntatically and semantically correct J73 program. The compiler shall not produce spurious warning or error messages and, when processing a correct J73 program, shall not produce target code that is syntatically or semantically incorrect or that does not correctly implement the J73 input program. When processing an input program that includes a single error, the compiler shall detect 100% of syntax errors and all detectable semantic errors. The compiler shall also indicate any capacity requirement that has been exceeded during processing. Warning and error messages shall be descriptive and shall indicate as precisely as possible the exact nature and location of the error. The detection of an error shall not cause the compiler to stop processing an input program unless the error is of such magnitude that no reasonable syntatic recovery action is possible. When this is the case, the compiler may, after issuing the appropriate messages, gracefully terminate processing. In other cases, the compiler may totally inhibit the generation of an object program if errors in its input prevent a reasonable interpretation and translation.

3.2.1.2.8 Compatibility. The operation of the compiler and the format of its inputs and outputs shall be totally compatible with the assembler and linkage editor described in this specification.

3.2.1.3 Compiler outputs. The outputs which shall be produced by the compiler are described in the following subparagraphs.

3.2.1.3.1 Object output. The compiler shall generate symbolic assembler language on 80 byte card image records as its objective output. This output shall be placed in the sequential or partitioned data set described by the JCL DD card //J73OBJ. If this data set is a partitioned data set, then the output shall be placed in the member name corresponding to the module name supplied to the compiler units input parameter string.

3.2.1.3.1.1 Relocatable output. The output code produced by the code generation component shall consist of an assembly language module which shall be partitioned into distinct relocatable submodules as required. Relocatable submodules shall be generated to contain unmodifiable instructions. A separate submodule shall be generated to contain unmodifiable data. These submodules shall be suitable for placement in read only memory. Other relocatable submodules shall contain data to be placed in read/write memory. When directed by its J73 source program, the compiler shall generate nonrelocatable read only and read/write submodules with specifically defined origin addresses. The compiler shall place only that data specified by its input program into these nonrelocatable submodules.

3.2.1.3.1.2 Annotation of symbolic output. Each line of assembler language code generated shall contain, when appropriate, right side comment field annotation of the J73 input statement number and J73 variable names applicable to that assembler statement. In addition, when directed by an input parameter or option, the actual J73 input statements shall be included in the output code in the form of assembler language

comments. Each J73 statement shall be placed immediately preceeding the bulk of the instructions generated to implement that statement. Implementation of any optimization performed by the compiler should be considered to take precedence over exact implementation of this requirement.

3.2.1.3.2 Symbolic debug output. Symbolic debug information generated by the compiler shall be placed in the sequential or partitioned data set described by the JCL DD card //J73DEBUG. If this data set is a partitioned data set, then the output shall be placed in the member name corresponding to the module name specified to the compiler on its input parameter string.

3.2.1.3.3 Printable output. All listing output from the compiler shall be directed to a single sequential or partitioned file described by the JCL DD card //J73LIST. If this card describes a partitioned data set, then the output shall be placed in the member name corresponding to the module name supplied to the compiler on its input parameter string.

3.2.1.3.3.1 Source listing. The compiler shall produce a listing of the source program input. Each input record shall be printed in full and should be prefaced by a compiler assigned statement number. When directed by an input parameter or option, the compiler shall also include in the source listing any referenced copy or symbolic compools. Each referenced file shall be included in the source listing at the point of its invocation. A separate input parameter or option shall cause the compiler to place diagnostic messages interspersed within the source listing. Each message shall preceed the source line which causes the generation of the message.

3.2.1.3.3.2 Message listing. The compiler shall produce a listing of error and diagnostic messages following the source listing. These messages shall contain references to the appropriate source input line numbers. All the various

messages produced by the compiler shall be numbered and classified into severity classes such as information, warning, error, severe error and fatal error. The compiler shall accept an input parameter or option which shall specify a message severity suppression level. Messages of severity level below this shall not be produced. When no messages are produced by the compiler, the generation of any message listing header pages shall be suppressed. The generation of the message listing is in addition to any messages interspersed in the source listing.

3.2.1.3.3.3 Environment listing. The compiler shall produce lexically sorted listing of all those user defined symbolic names referenced by the program. This listing shall include as a minimum, the declared attributes of each name along with the statement number (and compool module name, if applicable) of the names declaration, the scope of the declaration and the relative memory location of any storage implied by the declaration. Separate entries shall be made in this listing for identical names declared at different scope levels. Entries shall not be made for names declared in any invoked preprocessed compools unless the name is referenced (explicity or implicity) by the program.

3.2.1.3.3.4 Set/used listing. The compiler shall produce lexically sorted listing of all those user defined symbolic names referenced by the program. This listing shall provide, for each name, the origin and line number of the names declaration along with the line number of each reference to the name in the input program. Each reference should include, when applicable, an indication whether the reference to the name was to read or to set (or both) any storage allocated for the name. Separate entries shall be made in this listing for identical names declared at different scope levels. Entries shall not be made for names declared in any referenced preprocessed compools unless the name is referenced (explicitly or implicitly) by the program. This listing may be a separate listing produced by the compiler, or alternately, this information may be contained in the Environment listing.

3.2.1.3.3.5 Storage and run-time requirements listing. The compiler shall produce an output listing which provides the storage, stack frame and run time support requirements of each

procedure, function and block definition in the input program. Storage requirement shall be given in decimal number of words and shall be partitioned into the number of protected and the number of unprotected words required. Stack frame requirements shall be the number of words required for one call of each procedure or function, but shall not include the requirements of any (non-inline) procedures, functions, or run-time support called by that program. The run-time support requirements shall be given in the form of a list of run-time support modules required by the input programs and the line numbers of the input which caused the run-time support module to be invoked.

3.2.2 Assembler. The F-16 Integrated J73 Support Software System Assembler shall accept compiler generated and programmer generated target machine assembler language programs and shall generate relocatable and absolute object modules for input to the Support Software System linkage editor. A definition of the assembler language to be processed by the assembler is given in Appendix C to this specification. A definition of the object file output is provided in Appendix D. Each assembler developed under this specification shall be designed so that target machine pecularities and dependencies are isolated to a minimum number of modules and so that maximum use is made of a kennal set of target independent processing modules.

3.2.2.1 Assembler inputs. The assembler shall accept its inputs from external files supplied through the JCL stream. The assembler shall also accept a character string option list consisting of as many as 100 characters as an input parameter supplied through the JCL stream.

3.2.2.1.1 Source input. Input to the assembler shall consist of standard 80 character card image records containing statements written in the target machine symbolic assembler language. The assembler shall be capable of locating its source input from one of several alternative sources. If NNNN is the module name supplied in the input parameter string, then the assembler shall locate its input according to the following rules, applied in the precedence specified below.

a. From the JCL DD card //NNNN specifying a sequential file,

b. From member NNNN of the partitioned data set specified by the JCL DD card //ASMLIB,

or c. From the JCL DD card//SYSIN specifying a sequential file.

If the assembler cannot find its input in any of the above locations, it shall return to its caller with a severe error return code.

3.2.2.2.1.2 Macro input. The assembler shall be capable of locating any invoked macros from one of three alternative sources. If MMMM is the name of the macro invoked, then the assembler shall locate that file according to the following rules, applied in the precedence specified below.

a. From a macro definition of MMMM within the input source program,

b. From the JCL DD card //MMMM specifying a sequential file,

or c. From member MMMM of the partitioned data set specified by the JCL DD card //MACLIB.

3.2.2.1.3 Symbolic debug input. The assembler shall accept symbolic debug information produced by the compiler and, after further processing, pass this information to the linkage editor. If NNNN is the module name supplied in the input parameter string, then the assembler shall input this information from member NNNN of the partitioned data set specified by the JCL DD card //J73DEBUG. Because original assembly language programs will not have associated debug information, presence of this member, or the entire file shall be considered optional and its absence shall not be considered an error.

3.2.2.2 Assembler processing. The processing performed by the assembler shall include all the functions necessary to convert a target machine assembly language program into object code ready for input by the linkage editor.

3.2.2.2.1 Submodule processing. Each module input to the assembler shall consist of one or more submodules. Each submodule processed shall be identified as either relocatable or absolute. Macro definitions shall be known throughout the entire input program. Symbols shall not be known across submodule boundaries except as provided through the use of EXTERNAL symbol definitions. Each submodule shall be identified by the name of the module being processed plus one additional alphanumeric character provided by the input source program.

3.2.2.2.2 Provisions for memory protection. The assembler shall provide the directives or pseudo operations necessary to support memory protection in the target machine. Each word of object output by the assembler shall carry a protection attribute. Provisions shall be made for input programs to specify the protection attribute of words to be generated and to alter that attribute as the source inputs are processed. Provisions shall also be provided to specify the protection attribute of an entire submodule at the beginning of each submodule.

3.2.2.2.3 External symbol processing. The assembler shall provide the capability for input programs to specify symbols as being external to the current submodule being processed. The value of such symbols shall be resolved in the object code by the linkage editor. External symbols shall be allowed anywhere in the input that allows the use of internal symbols, such as in simple expressions. The equivalence of an internal symbol to an external symbol shall be processed correctly by the linkage editor. The internal symbol name shall not, however, be processed so as to conflict with the external symbol of the same name. The assembler shall load external fields of its object and print output with binary zeros.

3.2.2.2.4 Base register offset calculation. When the target machine instruction set architecture provides a base or index plus offset addressing mode, the assembler shall support the automatic calculation of operand address offsets. This capability shall be provided through the use of a psuedo-operation.

3.2.2.2.5 Psuedo operations. The assembler shall provide, but not be limited to, the psuedo operations listed in Table I-II.

TABLE I-II ASSEMBLER PSUEDO OPERATIONS

1. Title the current assembly
2. Start listing on a new page
3. Insert a specified number of spaces into listing
4. Enable list of source input
5. Disable listing of source input
6. Enable listing of expanded macro statements
7. Disable listing of expanded macro statements
8. Define memory words for data or address constants in character decimal or hexadecimal format with optional scaling
9. Declare symbol as external to this assembly
10. Declare a symbol in this assembly to have the external attribute
11. Define the value of a symbol, EQUATE
12. Declare a submodule as relocatable
13. Declare a submodule location as absolute and provide the origin
14. Enable memory protect
15. Disable memory protect
16. Define macro definition start and end
17. Set conditional assembly symbol
18. Test conditional assembly symbol

3.2.2.2.6 Macro capability. The assembler shall provide a macro instructions capability for repeating frequently referenced assembler instructions and pseudo operations. Macros shall be defined either in the input source program or in an external macro library. Macro instructions shall accept symbolic input parameters which may be used in the expansion of the instructions within the macro, and may also be used by conditional assembly test and set instructions within the macro.

3.2.2.2.7 Conditional assembly. The assembler shall provide the capabilty to conditionally generate object code from its input based upon the value of conditional assembly symbols. Provisions shall be made to set and test these symbols and to control the generation of code from the input program and its invoked macros based on the value of these symbols.

3.2.2.2.8 Symbolic debug support. The assembler shall produce, as part of its output, information to be used by the linkage editor to support symbolic debugging at the target machine level. Information from the J73 compiler shall be combined with information from the assembler to produce this output. Generation of this data shall be controlled by an input parameter or option. The exact content and form of this output is specified in Appendix F.

3.2.2.2.9 Error recognition and recovery. The assembler shall correctly process and produce target object code implementing any syntatically and semantically correct input program. The assembler shall not produce spurious warning or error messages, and when processing a correct input program, shall not produce target code which is invalid or which does not correclty implement the input program. When processing an input program that includes a single error, the assembler shall detect 100% of syntax errors and all detectable semantic errors. The assembler shall also indicate any capacity requirement that has been exceeded during processing. Warning and error messages shall be descriptive and shall indicate as precisely as possible the exact nature and location of the error. The detection of an error shall not cause the assembler to stop processing. Incorrect operand fields shall be

assembled as zero bits. Undefined instruction codes shall be assembled as one or more words of NOP instruction bit patterns. The number of these words included shall be equal to the number of words required to effect a transfer to any location in the machine address space.

3.2.2.2.10 Compatibility. The operation of the assembler and the format of its inputs and outputs shall be totally compatible with the compiler and linkage editor described in this specification.

3.2.2.3 Assembler outputs. The outputs which shall be produced by the assembler are described in the following subparagraphs.

3.2.2.3.1 Object output. The assembler shall generate relocatable and absolute object submodules for input to the Support Software System linkage editor. In order to support the linkage editor in the generation of an absolute program listing, the assembler shall include all of its input source records as part of its object file output. A definition of the object file output format is provided in Appendix D. The object output of the assembler shall be placed in the sequential or partitioned data set described by the JCL DD card //ASMOBJ. If this data set is partitioned, then the output shall be placed in the member name corresponding to the module name supplied to the assembler on its input parameter string.

3.2.2.3.2 Symbolic debug output. Symbolic debug information generated by the assembler (including information provided by the compiler) shall be placed in the sequential or partitioned data set described by the JCL DD card //ASMDEBUG. If this data set is a partitioned data set, then the output shall be placed in the member name corresponding to the module name specified to the assembler on its input parameter string.

3.2.2.3.3 Printable output. All listing output from the assembler shall be directed to a single sequential or partitioned file described by the JCL DD card //ASMLIST. If this card describes a partitioned data set, then the output shall be placed in the member name corresponding to the module name supplied to the assembler on its input parameter string.

3.2.2.3.3.1 Source listing. The assembler shall produce a listing of the source program input. This "side-by-side" listing shall provide the following information, printed in columns:

a. An assembler generated statement number.

b. The relocatable program address in hexadecimal.

c. The target computer instruction word or data word generated by the assembler. When more than one word of output is generated for a single output line, the higher address portions generated should be printed on succeeding lines, with the statement number and statement fields blank.

d. The full 80 column input source record.

Complete error messages shall be generated on a separate line immediately preceeding the statements in error. Statements generated as the result of a macro expansion shall be denoted in the listing. For all comment statements, only the statement number and 80 column statement field shall be printed. The card content of the source listing shall be controlled by the use of certain pseudo operations described in 3.2.2.2.5.

3.2.2.3.3.2 Message listing. The assembler shall produce a listing of error and diagnostic messages following the source listing. These messages shall contain references to the appropriate source input line numbers. All the various messages produced by the assembler shall be numbered and classified into severity classes such as information, warning, and error. The assembler shall accept an input parameter or option which shall specify a message severity suppression level. Messages of severity level below this shall not be produced. When no messages are produced by the assembler the generation of any message listing header pages shall be suppressed. The generation of the message listing is in addition to any messages interspersed in the source listing.

3.2.2.3.3.3 Cross reference listing. The assembler shall produce an lexically sorted listing off all those user defined symbols (including macro names) used by its input program.

This listing shall provide, for each name, the line number of symbol's definition, the value of the symbol (or an indication that the symbol is external or undefined) and the line number of each reference to the symbol in the input program. Separate entries shall be made for identical names declared in separate submodules.

3.2.2.3.3.4 Requirements listing. The assembler shall produce an output listing which provides the storage requirements of each submodule processed. The storage requirements shall be given for each submodule in decimal number of words and shall be broken into the number of protected and the number of unprotected words required.

3.2.3 Linkage editor. The F-16 Integrated Support Software System linkage editor shall merge separately assembled modules of an operational flight program and produce an absolute load module suitable for input to General Dynamics computer support equipment. Each linkage editor developed under this specification shall be designed so that target machine pecularities are isolated to a minimum number of modules and so that maximum use is made of a kennel set of target independent processing modules.

3.2.3.1 Linkage editor inputs. The linkage editor shall accept its inputs from external files supplied through the JCL stream. The linkage editor shall also accept a character string consisting of up to 100 characters as an input parameter supplied through the JCL stream.

3.2.3.1.1 Control input. The operation of the linkage editor shall be controlled by input control statements provided the file specified by the JCL DD card //LINKCNTL. The linkage editor shall accept control statements which perform the following functions.

a. Provide an operational flight program title to identify the listings and load modules to be produced,

b. Select a specific object module for inclusion in output load module,

c. Force a specified submodule to be placed in a specific memory location,

d. Force a specified submodule to assume a specific memory protection attribute,

and e. Provide the definition of the memory spaces available in the target machine and the protection status of these spaces if such status is to be predefined.

3.2.3.1.2 Object input. The linkage editor shall input the object modules to be processed from the partitioned data set defined by the JCL DD card //ASMOBJ. If no select statements are provided in the control input, then all members of this dat set shall be processed. If any member is specifically selected, then only those members which are selected shall be processed. A definition of the object file format is provided in Appendix D.

3.2.3.1.3 Library input. The linkage editor shall input object modules from an alternate input library in order to resolve unresolved external references. After all input modules have been processed, if EEEE is the name of an unresolved external reference, the linkage editor shall attempt to resolve this reference through the inclusion of member EEEE from the partitioned data set defined by the JCL DD card //LINKLIB. When a single member of this library defines more than one external symbol, the directory of this partitioned data set will contain "alias" entries for referring to this one member. When a member of the library is included in a load module, all external references which that module defines shall be resolved. No member of the library shall be included in a load module more than once.

3.2.3.1.4 Symbolic debug input. The linkage editor shall accept symbolic debug information produced by the assembler and, after further processing, shall output symbolic debug information for laboratory computer support equipment. The linkage editor shall retrieve and process this information only when directed by an input option or parameter to generate symbolic debug output. If NNNN is the name of a module included in the load module output (either by selection or

automatic inclusion), then the linkage editor shall input symbolic debug information from member NNNN of the partitioned data set specified by the JCL DD card //ASMDEBUG.

3.2.3.2 Linkage editor processing. The processing performed by the linkage editor shall include all the functions necessary to convert separately assembled modules into a single operational flight program load module. These functions shall include external reference resolution, the assignment of relocatable modules to absolute memory locations, and the management of memory protection assignments.

3.2.3.2.1 Memory assignment processing. The linkage editor shall allocate relocatable object submodules to specific memory addresses. The algorithm employed to accomplish this task shall require the approval of General Dynamics. This algorithm shall be designed to make maximum use of the memory locations available to the linkage editor (but not used by absolute object modules) and shall minimize the number of sections of unused locations in a fully loaded target machine. The definition of any predefined read only memory areas, and the implementation of the target machine memory protections mechanism shall be used by this algorithm during its processing. No actual memory words shall be allocated or required for the processing of object submodules which do not define or allocate any storage. Processing of this type of module shall not affect the memory assignments algorithm or the memory protection status of any words or blocks of words.

3.2.3.2.2 Memory protection provision. The linkage editor shall be designed to produce load modules for target machines with various memory protections mechanisms. As a minimum the linkage editor shall be capable of producing load modules for machines with the following types of memory, with no modification to the linkage editor source program:

a. Noncontiguous pre-defined areas of program read only, data read only and read/write memory

b. Read/write memory for which memory protection status is assigned by the linkage editor in blocks of N contiguous words

and c. Read/write memory for which memory protection status is assigned by the linkage editor for each word.

The type of memory for the target machine shall be defined by statements in the control input to the linkage editor.

If memory protections is being assigned on a block by block basis, then the linkage editor shall provide a means to build protection assignment information into the target computer load module at the location defined by a special reserved external symbol. In addition, if requested by an input parameter or control statement this same information shall be encoded at the end of the load module output file.

3.2.3.2.3 Memory checksum provisions. The linkage editor shall compute a checksum of all the protected memory in the target computer load module. The exact algorithm to be used to compute this checksum shall be defined by General Dynamics. This checksum shall be built into the target computer load module at the location defined by a special reserved external symbol. If also requested by an input parameter or control statement, this same information shall be encoded at the end of the load module output file.

3.2.3.2.4 Symbolic debug support. The linkage editor shall produce, as part of its output, information to be provided to laboratory computer support equipment to support the symbolic debugging of operational flight programs loaded into their target machines. Information produced by the compiler and assembler shall be input by the linkage editor, correlated with the actual load module produced, and output to an external file. The exact content and form of this output is specified in Appendix F.

3.2.3.2.5 Partial load capability. During normal operation, the linkage editor shall produce an output load module which loads the contents of all memory locations which are defined in its control input file. Unused locations shall be loaded with a TBD constant and shall be given the protected memory protection attribute. Optionally when directed by an input parameter or control statement, the linkage editor shall produce a load module which only loads those locations specified by its input object modules. The output format for these partial load files shall be such that several files may

be sequentially loaded by laboratory equipment to complete a single target program. When producing a partial load file, the linkage editor shall calculate the checksum assuming unloaded locations contain zero.

3.2.3.2.6 Error recognition and recovery. The linkage editor shall correctly process and produce a load module output program implementing any set of syntatically and semantically correct input object programs. The linkage editor shall not produce spurious warning or error messages, and when processing a correct set of input programs, shall not produce a load module which is invalid or which does not correctly implement the input programs. The linkage editor shall detect any syntax errors in its input control file, and any errors in the definition and use of external symbols in its object input. The linkage editor shall also detect any errors in the allocation and protection of memory. Such errors detected shall include the overlapping allocation of absolute submodules, the allocation of both protected and unprotected submodules to the same memory protection block and the allocation of a protected or unprotected submodule in violation of predefined memory attributes. The linkage editor shall also indicate any capacity requirements that has been exceeded during processing. It shall indicate when insufficiet memory has been allocated to accommodate all input object modules. The linkage editor shall be capable of processing object modules produced by the assembler from an input which contained errors. The detection of any error shall not cause the linkage editor to stop processing. Unresolved external fields shall be set to binary zeros. The content of multiple defined locations may be undefined.

3.2.3.2.7 Compatibility. The operation of the linkage editor and the formal of its inputs and outputs shall be totally compatible for use with the compiler and assembler described in this specification.

3.2.3.3 Linkage editor outputs. The outputs which shall be produced by the linkage editor are described in the following subparagraphs.

3.2.3.3.1 Load file output. The linkage editor shall produce a load module output file which shall represent the memory contents, memory protection status and memory checksum of a loaded target computer. This file shall also contain a eight bit ASCII representation of the program title supplied to the linkage editor. While normal medium for this load file will be nine track magnetic tape, in order to meet the requirements of General Dynamics' customer, this same file shall also be capable of being placed on eight channel punched tape. (installation dependent host software to actually transfer the file contents to punched tape will be provided by General Dynamics). In order to support this requirement, the load file format shall have the following characteristics:

a. Each byte of the load file shall carry its own internal (odd) parity bit,

b. The load file shall be divided into distinct, separately loadable sections. Each section, when punched, shall fit on a single 6 inch punched tape reel. (This is approximately 8k of 16 bit words if three load bytes are used to represent 16 bits),

c. Each section shall be preceeded by leader, a leader message, and the program title which shall be readable by maintenance personnel when physically inspecting the punched tape medium. The program title, a trailer message, and leader shall terminate the file. Both leader and trailer messages shall contain the section sequence number, such as "TAPE 3 OF 4 TAPES".

d. The installation dependent tape punching program must separately acess each load section. To support this requirement, each section shall be written as a single member of the load file partitioned data set defined by the JCL DD card //LOADFILE. (General Dynamics will concatinate all load sections to create a single sequential file for magnetic tape).

3.2.3.3.2 Symbolic debug output. Symbolic debug information generated by the linkage editor shall be placed in the sequential data set described by the JCL DD card //DEBUG.

3.2.3.3.3 <u>Printable output</u>. Except as noted in the following subparagraphs, all listing output by the linkage editor shall be directed to a single sequential data set described by the JCL DD card //LINKLIST.

3.2.3.3.3.1 <u>Selected module listing</u>. The linkage editor shall produce a listing of the names of all modules included in the output load module. This listing shall provide an indication of those modules which were included in the output from the library input file in order to resolve external references. This listing shall be printed in the order that the modules are selected.

3.2.3.3.3.2 <u>Memory allocation map</u>. The linkage editor shall produce a listing of all the submodules included in the output load module. This listing shall provide the following information, printed in columns:

a. Name of the submodule,

b. The starting and ending memory load address for the submodule (in hexadecimal),

c. If memory protection is not on a word basis, the protection status of the submodule,

d. An indication that the submodule is overlayed with other submodules,

e. The length, in decimal, of the storage allocated to the submodule,

and f. A list (in several columns, as space provides) of the external symbols defined by the submodule and the value of each of these symbols,

This listing shall be printed in the order of the address of each submodule. A blank line should be printed between submodule entries to enhance readability of the printout. Entries for submodules which do not allocate memory (and hence have no starting address) shall be placed at the end of the listing.

3.2.3.3.3.3 External symbol listing. The linkage editor shall produce an alphanumeric sorted listing of all external symbols defined in the output load module. This listing shall contain the symbol name, the name of the defining submodule and the value of the symbol in both hexadecimal and decimal.

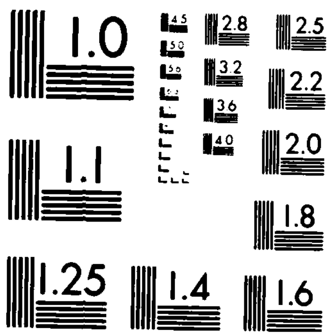
3.2.3.3.3.4 Memory dump. The linkage editor shall produce a memory dump style listing of the entire contents of the target computer memory as loaded by the output load file. This listing shall contain on each line an absolute memory address, and the contents of the next 16 memory words starting with the printed address. Three blanks shall separate the absolute address field from the memory word field. One blank shall separate each memory word from the other, with an extra blank inserted between the eighth and ninth memory words. When group of two or more lines that are to be printed are identical (except for the absolute address field) only the first line of the group will be printed; the next line will contain the message "Succeeding lines are identical." The group of identical lines will be terminated when the next line to be printed is not identical to the previous one. All addresses and memory words shall be represented by hexadecimal digits.

3.2.3.3.3.5 Absolute assembly listing. For each assembly language module processed by the linkage editor, an absolute assembly language listing shall be produced. This listing shall be identical in format to the assembly source listing produced by the assembler (see 3.2.2.3.3.1), except that the address column shall contain absolute memory addresses rather than relocatable address offsets. As with the assembly listing the exact content of this listing shall be controlled by certain listing control pseudo operations. If NNNN is the name of an assembly language module processed, then the absolute assembly listing for that module shall be placed in member NNNN of the partitioned data set described by the JCL DD card //ABSLIST.

3.2.3.3.3.6 Message listing. The linkage editor shall produce a listing of error and diagnostic messages. Warning and error messages shall be descriptive and shall indicate as precisely as possible the exact nature and location of the error. Such messages shall include the names of any (1) undefined or multiply defined external symbols, (2) submodules

AD-A150 584 PROCEEDINGS OF THE TECHNICAL FORUM (3RD) ON THE F-16 MIL-STD-1750A MICROP..(U) AERONAUTICAL SYSTEMS DIV WRIGHT-PATTERSON AFB OH J L PESLER ET AL. 06 MAY 82

UNCLASSIFIED ASD-TR-82-5011-VOL-2 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

whose memory assignment conflicts with other modules and (3) submodules that violates memory protection rules. For undefined external symbols, a message shall be issued for each occurrence of the symbol, stating the module name and target load location of the symbol. An error message shall also be issued when the value of the external symbol will not fit in the number of bits allocated in an occurrence of its use. All the various messages produced by the linkage editor shall be numbered and classified into several classes such as information, warning, error, severe error and fatal error. The linkage editor shall accept an input parameter or option which shall specify a message severity suppression level. Messages of severity level below this shall not be produced. When no messages are produced by the linkage editor, the generation of any message listing header pages shall be surpressed.

3.2.4. Special requirements. This paragraph specifies additional requirements for the Support Software System not provided in other sections of the specification.

3.2.4.1 Human performance. This paragraph is not applicable to the specification.

3.2.4.2 Government-furnished property list. This specification does not require any government-furnished computer programs. Any government furnished computer programs incorporated into the Support Software System (with or without modification) shall be clearly identified to General Dynamics.

3.2.4.3 Common print format. Each listing produced by a Support Software System program shall include one or more header lines at the top of each page. This header shall identify the title of the listing and the Support Software System program name and version which produced the listing. This listing shall also include the date and time the listing was produced (obtained from the host operating system) and a page number. Page numbers may sequentially increase throughout the entire print output of the program or may be reset at the start of each separate listing. For compilations and assemblies, the header shall contain the name of the program unit (Obtained from the input parameter string). For link edit listings, the header shall contain the title of the load module (obtained from the input control file). For absolute assembly listings, both the name of the module and the title of the program shall be printed. All listings references (produced by an Support Software System program) which refer to a basic memory word and/or address shall be formatted for printing as hexadecimal digits.

3.2.4.4 Efficient design of software. All programs of the Support Software System shall be designed and in a modular, structured fashion. Code shall be identified to show program structure and nesting levels and shall contain in-line comments to annotate the processes being performed. The generation of compact, highly efficient target code by the compiler must take precedence over execution time requirements on the host computer. However, General Dynamics expects all Support Software System programs to be reasonably efficient on the host system. The execution time of Support Software system programs should fall within the following guidelines:

a.	Compilation of a 250 line program, within	7.0 sec
b.	Preprocessing a 6000 line compool, 2800 symbols declared, within	45.0 sec
c.	Assemble program from (1) above, within	2.0 sec
d.	Link edit and generate absolute listing for a program requiring 25k (16 bit) words, within	180.0 sec

Times listed are CPU execution times on an IBM 3033 multiprocessor installed at General Dynamics, Fort Worth Division.

3.2.4.5 Implementation of new target machine code. All programs developed for the Support Software System shall be designed and documented so that the experienced personnel of General Dynamics can, with a reasonable effort, modify the programs provided in order to generate, assemble and link code for a new target machine instruction set architecture. Contractor shall use a clean, well documented interface between the compiler target independent component and code generation component of the compiler, and shall modularize the design of the assembler and linkage editor so that target dependencies are located in a minimum number of modules. The product specification for each program shall provide as an appendix, a guide for the retargeting of that program.

3.2.4.6 Rehostability. All programs developed for the Support Software System shall be modularized so as to minimize host computer dependencies. By providing individual replacements for host dependent modules, the entire Support Software System shall be rehostable on various machines of different architecture than the IBM 370. Programming constructs dependent on host computer word size shall be avoided.

3.3 Adaption. This paragraph specifies the requirements for the adaptation of this Support Software System for its use at specific installations. Additional adaptation requirements are specified in 3.2.4.

3.3.1 General environment. The Support Software System shall operate within the general environments specified in 3.1.1.

3.3.2 System parameters. The contractor shall identify all those system parameters required for each program of the Support Software System. These parameters shall include the amount of virtual storage required and, a description of all temporary files required, including estimated direct access space requirements. The contractor shall identify any other system parameters required. Contractor shall also furnish a description of a minimum configured system required for the Support Software System.

3.3.3 System capacities. The Support Software System shall provide the following minimum capacities.

3.3.3.1 Compiler capacities. The compiler shall provide the following minimum capacities when provided sufficient virtual storage:

Number of unique names	15000 names
Number of compools invoked	20 files
Number of copy files invoked	100 files
Maximum "copy file" nesting	10 levels
Number of "define" declarations	500
Number of statements and declarations	9999
Maximum nesting level	50 levels
Maximum character size	255 characters
Maximum size of output module	65536 words

Incorporation of capacities in addition to those listed shall require the approval of General Dynamics.

The compiler shall incorporate the following J73 implementation parameters:

INTPRECISION	15
FLOATPRECISION	23
FIXEDPRECISION	15
MAXINTSIZE	31
MAXFLOATPRECISION	39
MAXFIXEDPRECISION	31
MAXBYTES	255
MAXBITS	2048
MAXTABLESIZE	32767

3.3.3.2 Assembler capacities. The assembler shall provide the following minimum capacities when provided sufficient virtual storage:

Number of unique names	15000 names
Number of macro definitions	500
Maximum nesting of macro expansions	50
Number of input statements	50000
Number of external symbols referenced	500
Number of external symbols defined	500
Maximum size of output program	65536 words
Number of submodules per module	36

Incorporation of capacities in addition to those listed shall require the approval of General Dynamics.

3.3.3.3 Linkage editor capacities. The linkage editor shall provide the following minimum capacities when provided sufficient virtual storage:

Number of external names	15000 names
Number of included modules	500 modules
Number of submodules	2000 submodules
Maximum size of output program	524288 words

Incorporation of capacities in addition to those listed shall require the approval of General Dynamics.

4. QUALITY ASSURANCE PROVISIONS

4.1 Introduction. Development and testing of the F-16 Integrated Jovial J73 Support Software System shall be of high quality, adhering to the specific requirements set forth in the following paragraphs. Test plans and test procedures shall be developed to insure compliance with these requirements. Testing shall be performed in accordance with the plans and procedures. Functional tests shall be performed in addition to tests covering specific computer program components.

Each functional capability and performance requirement specified in Section 3 of this specification has a verification method specified in this section, as summarized in tabular cross references form in Table I-III. Any requirements or functional characteristics for which tests are not now defined shall be added by the contractor following the format and intent provided in this section. Methods of verification which shall be used in the accomplishment of the requirements of this section are:

a. Inspection. Inspection is defined as a visual verification that this CPCI as produced conforms to the documented requirement to which it was designed. Inspection of storage maps and program lists may be used as a means to verify total storage utilization, table sizes, and other aspects of this Support Software System which are either directly evident from examination of the compiled code or easily derived from the compiled code. Inspection may also be used as a means to verify aspects of program construction and documentation through examination of flowcharts and compiled code.

b. Review of analytic data. Analysis is defined as verification that a specification requirement has been met by technical evaluation of equations, charts, reduced data and/or representative data.

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Review of Analytic Data 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3. Requirements	X					--
3.1 Computer program definition		X				FQT
3.1.1 Interface requirements		X				FQT
3.1.1.1 Interface block diagram		X				FQT
3.1.1.2 Detailed interface definition		X				FQT
3.1.1.2.1 Host invocation of program					X	PQT
3.1.1.2.2 Input/output IBM interface				X		CPT & E
3.2 Detailed functional requirements	X					
3.2.1 J73 compiler		X		X	X	FQT
3.2.1.1 Compiler inputs		X		X		CPT & E
3.2.1.1.1 Source input					X	PQT
3.2.1.1.2 Compool data file input		X			X	PQT
3.2.1.1.3 Copy file input					X	PQT
3.2.1.2 Processing		X				PQT

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Review of Analytic Data 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.1 Target independent component		X				CPT & E
3.2.1.2.2 Code generator component		X		X		PQT
3.2.1.2.3 Run-time support requirements		X		X		PQT
3.2.1.2.4 Literal area		X	X			PQT
3.2.1.2.5 Machine-dependent subroutines		X	X			CPT & E
3.2.1.2.6 Optimizations for efficient code			X			PQT
3.2.1.2.6.1 Register, subroutine and parameters conventions		X	X			FQT
3.2.1.2.6.2 Abort statement processing					X	FQT
3.2.1.2.6.3 Nested procedure processing					X	FQT
3.2.1.2.6.4 Additional subroutine call optimization			X			PQT
3.2.1.2.7 Symbolic debug support		X	X			FQT
3.2.1.2.8 Preprocessed compools				X		FQT

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable   2 = Review of Analytic Data   4 = Test
I = Inspection   3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3.2.1.2.9 Error processing and recovery				X	X	CPT & E
3.2.1.2.10 Compatibility		X		X		FQT
3.2.1.3 Compiler outputs	X					
3.2.1.3.1 Object output					X	PQT
3.2.1.3.1.1 Relocatable					X	PQT
3.2.1.3.1.2 Annotation of symbolic output			X			PQT
3.2.1.3.2 Symbolic debug output					X	PQT
3.2.1.3.3 Printable output					X	PQT
3.2.1.3.3.1 Source listing			X			PQT
3.2.1.3.3.2 Message listing			X		X	FQT
3.2.1.3.3.3 Environment listing			X		X	FQT
3.2.1.3.3.4 Set/used listing			X		X	FQT
3.2.1.3.3.5 Storage and run-time requirements listing			X		X	FQT
3.2.2 Assembler		X		X	X	FQT

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Review of Analytic Data 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3.2.2.1 Assembler inputs		X		X	X	PQT
3.2.2.1.1 Source input					X	PQT
3.2.2.1.2 Macro input					X	PQT
3.2.2.1.3 Symbolic debug input					X	FQT
3.2.2.2 Assembler processing		X				FQT
3.2.2.2.1 Submodule processing		X			X	PQT
3.2.2.2.2 Provisions for memory protection		X				PQT
3.2.2.2.3 External symbol processing					X	FQT
3.2.2.2.4 Base register offset calculation					X	FQT
3.2.2.2.5 Psuedo operations					X	FQT
3.2.2.2.6 Macro capability				X		CPT & E
3.2.2.2.7 Conditional assembly					X	PQT

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Review of Analytic Data 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3.2.2.2.8 Symbolic debug support		X			X	FQT
3.2.2.2.9 Error processing and recovery				X	X	FQT
3.2.2.2.10 Compatibility				X	X	CPT & E
3.2.2.3 Assembler outputs	X					--
3.2.2.3.1 Object output		X			X	FQT
3.2.2.3.2 Symbolic debug output					X	CPT & E
3.2.2.3.3 Printable output					X	CPT & E
3.2.2.3.3.1 Source listing			X	X		FQT
3.2.2.3.3.2 Message listing			X		X	CPT & E
3.2.2.3.3.3 Cross reference listing					X	FQT
3.2.2.3.3.4 Requirements listing			X			CPT & E
3.2.3 Linkage editor		X		X	X	FQT

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Review of Analytic Data 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3.2.3.1 Linkage editor inputs		X		X		PQT
3.2.3.1.1 Control input					X	PQT
3.2.3.1.2 Object input					X	PQT
3.2.3.1.3 Library input					X	PQT
3.2.3.1.4 Symbolic debug input					X	FQT
3.2.3.2 Linkage editor processing		X				FQT
3.2.3.2.1 Memory assignment processing				X	X	FQT
3.2.3.2.2 Memory protection provisions		X		X	X	CPT & E
3.2.3.2.3 Memory checksum provisions					X	CPT & E
3.2.3.2.4 Symbolic debug support		X		X		FQT
3.2.3.2.5 Partial load capability					X	PQT
3.2.3.2.6 Error processing and recovery					X	FQT
3.2.3.2.7 Compatibility		X				CPT & E

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable   2 = Review of Analytic Data   4 = Test
1 = Inspection   3 = Demonstration

Section 3 Requirement Reference	Verification Methods NA	1	2	3	4	Section 4 Verification Requirements
3.2.3.3 Linkage editor outputs	X					--
3.2.3.3.1 Load file output		X		X		FQT
3.2.3.3.2 Symbolic debug output					X	CPT & E
3.2.3.3.3 Printable output					X	CPT & E
3.2.3.3.3.1 Selected module listing				X		CPT & E
3.2.3.3.3.2 Memory allocation map			X		X	CPT & E
3.2.3.3.3.3 External symbol listing			X			CPT & E
3.2.3.3.3.4 Memory dump			X			CPT & E
3.2.3.3.3.5 Absolute memory listing			X		X	FQT
3.2.3.3.3.6 Message listing				X		CPT & E
3.2.4 Special requirements	X					--
3.2.4.1 Human performance	X					--

TABLE I-III

VERIFICATION CROSS REFERENCE INDEX

NA = Not Applicable 2 = Review of Analytic Data 4 = Test
1 = Inspection 3 = Demonstration

Section 3 Requirement Reference	NA	1	2	3	4	Section 4 Verification Requirements
3.2.4.2 Government furnished property list	X					--
3.2.4.3 Common print format			X	X		PQT
3.2.4.4 Efficient design of software				X		FQT
3.2.4.5 Implementation of new target machine code		X				CPT & E
3.2.4.6 Rehostability		X				CPT & E
3.3 Adaption	X					--
3.3.1 General environment				X		FQT
3.3.2 System parameters				X		FQT
3.3.3 system capacities	X					--
3.3.3.1 Compiler capacities					X	CPT & E
3.3.3.2 Assembler capacities					X	CPT & E
3.3.3.3 Linkage editor capacities					X	CPT & E

c. Demonstration Demonstration is defined as an uninstrumented test where success is determined by observation. Included in this category are tests that require simple quantitative measurements such as dimensions, time to perform tasks, and ability to restart.

d. Test. Test is defined as verification that a specification requirement is met by a thorough exercising of the applicable element under appropriate conditions in accordance with test procedures. Verification by test includes aspects verifiable at the time the test is conducted as well as aspects which can be verified through subsequent review of test data.

As a part of the test plan the contractor shall specify when and how verification will be accomplished, shall notify General Dynamics of the times and locations where tests and demonstrations will be conducted, shall make provisions for General Dynamics attendances at such tests and demonstrations, and shall make available the program data, test results and other material necessary to ensure that successful verification has been accomplished.

4.1.1 Category I test. The Category I testing of the F-16 Integrated Jovial J73 Support Software System shall be subdivided into the following types:

a. Computer programming test and evaluation - 4.1.2

b. Preliminary qualification test ----------- 4.1.3

c. Formal qualification test --------------- 4.1.4

The Category I testing shall include testing at progressive levels of Support Software System integration to verify the successful operation of this Support Software System. This testing shall be performed on individual subroutines, on groups of modules, and on the entire Support Software System. Test shall also be performed to verify that the module interfaces and program linkages are correct. Formal qualification testing of this Support Software System shall be performed using a host computer environment as described in 3.1.1. Verification of this Support Software System shall include demonstrations and tests of this Support Software System in actual operation.

4.1.2 Computer programming test and evaluation. Computer programming test and evaluation shall be conducted as an integral part of the development process and shall be primarily directed toward the verification of CPCs, collections of related CPCs, and interfaces between computer subprograms, equipments, and systems. These tests shall be conducted prior to and in parallel with preliminary or formal qualificiaton tests. Approved test procedures are not required for verifications conducted under this paragraph. Test documentation in contractor format is acceptable providing the following minimum data is maintained for evaluation: (1) component identification, date of verification, method of verification, programmer's name, (2) exact listing of component as successfully tested, and (3) test description giving inputs, outputs, and approach. The Section 3 requirements to be verified under this paragraph are specified in Table I-III.

4.1.3. Preliminary qualification tests. Prior to integrated testing of the complete support, preliminary qualification tests oriented toward verifying proper performance of certain critical portions of the support shall be accomplished. Preliminary qualification testing shall be directed toward verification of subprograms which are either (1) essentially I/O-independent; that is, are primarily logical components operating on a data base, or (2) involve only interaction with other parts of the system resulting in a functional capability of limited size. Preliminary qualification tests shall be accomplished in accordance with approved test procedures. Preliminary or Formal Qualification tests shall be used to verify each of the requirements of paragraph 3 as indicated by Table III.

4.1.4 Formal qualification tests. The functional requriements of this CPCI specified in Section 3 and not verified under 4.1.2, 4.1.3, or intended for verification in Category II testing shall be verified through formal qualification tests. Testing conducted under this paragraph shall be directed toward verification of performance requirements which rely on the satisfactory operation of the integrated Support Software System. The requirements to be verified through formal qualificatoin testing and the method of verification to be used are as specified in Table I-III.

4.1.5 Category II system test. Category II test is not required for the F-16 Integrated Jovial J73 Suppórt Software System.

4.2 Test requirements. Computer Programming Test and Evaluation testing shall be conducted during the development stages of the Support Software System and will test the computer subprograms, programs and their integration. These tests will support the design and development of the computer programs and will serve to check the compatibility of computer program code and design. These testing activities shall continue throughout the entire development phase. Testing and development shall be synchronized in such a way as to allow each subprogram and program to be tested as early as possible during implementation and to allow early verification of proper system integration.

In addition to the CPT&E testing, the support software system shall successfully pass the PQT and FQT tests as defined in Table III as further refined by the following subparagraphs.

4.2.1 Implementation standards. The Support Software System shall be tested to assure that it is implemented in such a manner that:

a. All code is indented to clearly denote logical levels of constructs.

b. All code has sufficient embedded annotation to explain inputs, outputs, and other aspects not obvious in the code itself.

c. The documentation provided correlates properly and easily with the generated code.

4.2.2 Acceptability of test plan. The test plan requested in this section must meet the requirements of this section.

It shall include the accomplishment of the activities summarized by Table I-III and any required revisions.

To the extent that portions of the Support Software System prepared for the target computer are identical to portions of the system prepared for other target computers, it is not necessary that tests of those common portions be duplicated for each support software/target computer combination. It is, however, necessary that all portions of the software be tested and that integrated total function tests be performed for each such combination.

The test plan shall require complete execution of all verification activities at the contractors facility.

4.2.3 Accomplishment of tests. The verification activities required by the test plan shall be successfuly accomplished and recorded for later review to the satisfaction of General Dynamics.

4.2.4 Efficiency and completeness of system. The Support Software System for each specified target computer shall be capable of generating all of the instructions of the target computer. Code generated by the compiler shall not be more than 15% inefficient in terms of memory utilization as compared with hand compilation of the same J73 source statements by an experienced programmer. Test cases to verify this requirement will be provided at time of acceptions by General Dynamics.

4.2.5 Usability. The Support Software System must operate correctly and be usable in a straight-forward manner using General Dynamics facilities and host computers. The contractor shall specify, subject to General Dynamics concurrence, a minimal subject of the tests suitable for demonstrating this requirement.

4.2.6 General Dynamics provided tests. The compiler (and entire Support Software System) shall properly process any legal combination of J73 constructs, assembler and linker input statements, and input directions. General Dynamics reserves the right to generate additional test cases as required to verify the acceptability of this requirement.

4.2.7 Operability. The Support Software System shall conform to the language specifications of MIL-STD-1589 and other requirements as specified in Section 3. Specific verifications to be made include, but are not limited to those listed in the following subparagraphs.

4.2.7.1 Language conformance.

a. Verify that legal statements are accepted by the compiler under test, and that the user-readable information produced by the compiler is correct. These tests are not intended to verify that the actual translation to machine language is correct.

b. Verify that the compiler under test rejects statements which are not legal and that appropriate diagnostic messages are produced when such statements are recognized.

c. Verify the correct operation of compiler directives as defined in the language specification.

d. Check that capacity requirements and constraints are met.

e. Verify that correct programs are translated into correct object code.

4.2.7.2 <u>Assembler and linker tests</u>. Verify proper operation during:

a. Processing of symbolic opcodes.

b. Processing of symbolic and absolute internal addresses.

c. Processing of external symbolic addresses.

d. Allocation of memory during the assembly and linking processes.

e. Detection of error and warning conditions and issuance of appropriate message.

f. Generation of listings, error tables and cross reference tables.

g. Generation of internal and external symbol tables, module tables, memory tapes, object cards and object tape image files.

h. Generation of object data.

i. Memory allocation of absolute and relocatable modules.

j. Memory allocation of protected and unprotected memory.

4.2.7.3 <u>Option control tests</u>. Testing shall include suppression and activation of the following options:

a. Set-used and environment listing

b. Source code listing

c. Object code listing

d. Error suppression

e. COPY expansions

f. Lines per page option

g. Preprocess compool mode

h. Syntax checking mode

i. Assembly language object output

4.3 Acceptance test requirements. This paragraph is not applicable to this specification.

5. PREPARATION FOR DELIVERY

This section is not applicable to this specification.

6. NOTES

None

APPENDIX A

DEFINITION OF IMPLEMENTATION AND TARGET COMPUTER DEPENDENT FEATURES OF MIL-STD-1589

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering the appendix as part of the completed Computer Program Development Specification.

APPENDIX B

TARGET - INDEPENDENT/CODE GENERATOR INTERFACE DEFINITION AND INTERMEDIATE LANGUAGE DESCRIPTION

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering this appendix as part of the completed Computer Program Development Specification.

APPENDIX C

SYMBOLIC ASSEMBLY LANGUAGE

DEFINITION FOR TARGET COMPUTER

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering this appendix as part of the completed Computer Program Development Specification.

APPENDIX D

OBJECT FILE FORMAT DEFINITION

FOR TARGET COMPUTER

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering this appendix as part of the completed Computer Program Development Specification.

APPENDIX E

LOAD FILE FORMAT DESCRIPTION

FOR TARGET COMPUTER

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering this appendix as part of the completed Computer Program Development Specification

APPENDIX F

SYMBOLIC DEBUG OUTPUT DESCRIPTION

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering this appendix as part of the completed Computer Program Development Specification.

APPENDIX G

DESCRIPTION OF TARGET COMPUTER
RUN-TIME MEMORY ALLOCATION, REGISTER USAGE, AND SUBROUTINE LINKAGE CONVENTIONS

The contents of this appendix shall be determined by the developer. The developer shall be responsible for delivering this appendix as part of the completed Computer Program Development Specification.

DUAL MIL-STD 1750A ASSEMBLY SYNTAX

FOR

F-16A+ INTEGRATED SUPPORT SOFTWARE SYSTEM

APRIL 1, 1982

SUBMITTED TO:

GENERAL DYNAMICS
FORT WORTH, TEXAS

CONTRACT NO. 960829

SUBMITTED BY:

PROPRIETARY SOFTWARE SYSTEMS ,INC.
9911 WEST PICO BOULEVARD
PENTHOUSE K
LOS ANGELES, CALIFORNIA 90035
(213)553-2997

ASD 82-0891

DUAL SYNTAX FOR MIL-STD 1750A ASSEMBLY LANGUAGE PROGRAMS

1 - INTRODUCTION

This document describes the DUAL syntax for writing MIL-STD 1750A (M1750A) assembly language programs. The MIL-STD 1750A DUAL cross assembler and linkage editor translate source modules into MIL-STD 1750A executable load modules.

The 1750A DUAL cross assembler consists of the complete 1750A instruction set, several 1750A specific assembler directives, and all of the standard DUAL assembly/macro language facilities. The following sections describe the new directives, the 1750A instruction syntax, and the addressing modes. The DUAL User's Manual provides a detailed description of the assembler directives, functions, constants, meta (macro) facilities, and linkage editor.

2 - REGISTERS

There are 16 general purpose registers in the M1750A computer. These registers are denoted by the built-in symbols R0,R1,...,R15. It should be noted that numbers cannot denote registers; however, the DUAL EQU directive can be used to define a new symbol to represent a register.

For example, if register R15 is to be used as a stack pointer, then the following directive defines a more appropriate name:

```
SP   EQU   R15
```

After this directive, both symbols SP and R15 can be used to denote register number 15.

Symbols RH0,RH1,...,RH15 are defined to denote the higher byte of registers R0,...,R15, and symbols RL0,RL1,...,RL15 denote the lower byte of registers R0,...,R15. These byte registers are only used with load byte (LDB) and store byte (STB) instructions.

3 - GENERAL INSTRUCTION FORMAT

The general format for 1750A instructions is:

```
[LABEL]   COMMAND   ARGUMENTS [COMMENTS]
```

The label field may be void or any legal DUAL symbol. The COMMAND field must be an operation code mnemonic (Sec. 5), a DUAL META reference (see DUAL Users Manual), or a DUAL directive.

The argument field consists of zero or more subfields. The subfields are separated by commas. Each subfield format depends on the instruction's addressing modes (Sec. 4). The comment field is any string of characters.

4 - ADDRESSING MODES

The number of arguments depends on the actual instruction and the type of argument utilized. The different argument types are referred to as addressing modes. The syntax for each addressing mode is described below.

4.1 - REGISTER (R)

Format: Rn

Description: Rn can either be one of the built-in symbols R0, R1,...,R15, or a user defined symbol which is set to one of the above symbols by a SET directive. The LDB and STB instructions require RB0...RB15 instead of R0...R15.

Example:

LABEL	COMMAND	ARGUMENT
	LD	R2,R1

This instruction has two arguments, and both are in the register mode. The instruction specifies loading R1 with the contents of R2.

4.2 - MEMORY DIRECT (D)

Format: ADDR

Description: The argument consists of a symbol which denotes a memory address. The ADDR must appear in the label field of an instruction, in a data generation directive, or have been equated to another address, or as the argument field of a REFER directive (i.e. defined in a separate module).

Example:

LABEL	COMMAND	ARGUMENTS
X	DATA	0
	ADD	X,R1

The second argument is in memory direct mode. It means add the contents of memory location X to the contents of R1 and store the result in R1.

4.3 - MEMORY DIRECT INDEXED (DX)

Format: ADDR(RX)

Description: The ADDR is as defined in Sec. 4.2. The index register RX can be any of the registers R1,R2,...,R15. RO is not a legal index register.

Example:

```
LABEL     COMMAND   ARGUMENT
LBL       DATA      10
          LD        LBL(R1)
```

The second instruction's argument is in DX mode.

4.4 - MEMORY INDIRECT (I)

Format: @ADDR

Description: The ADDR is as defined in Sec. 4.2.

Example:

```
LABEL     COMMAND   ARGUMENT
LBL       DATA      0
          BR        @LBL
```

It means branch to the memory location whose address is in memory location LBL.

4.5 - MEMORY INDIRECT WITH PRE-INDEXING (IX)

Format: @ADDR(RX)

Description: The ADDR(RX) is as defined in section 4.3.

Example:

```
LABEL     COMMAND   ARGUMENTS
LBL       DATA      0
          BR        @LBL(R1)
```

It means branch to the memory location whose address is in memory location (LBL plus the contents of R1).

4.6 - IMMEDIATE (IM)

Format: #N or
##DATA

Description: If the initial character in the argument field is a `#`, then it is classified as an immediate short data, and if it is '##', then it is classified as full word immediate data.

Example:

```
LABEL     COMMAND   ARGUMENTS
          SETI      #0,R1
```

The first argument of the instruction is in immediate addressing mode. It means set the first bit of register R1 to one.

4.7 - BASE RELATIVE (B)

Format: DSPL(BR)

Description: DSPL is a constant, and the base register (BR) can only be one of the registers R12,...,R15.

Example:

LABEL	COMMAND	ARGUMENTS
	ADD	50(R13),R2

This second instruction means add the contents of the designated memory location (whose address is in 50 plus the contents of R13) to R2.

4.8 - BASE RELATIVE INDEXED (BX)

Format: BR(RX)

Description: Base register BR can only be one of the registers R12,...,R15.

Example:

LABEL	COMMANDS	ARGUMENTS
	ADD	(R12,R1),R2

This means add the contents of the designated memory location (whose address is the value in R12 + the value in R1) to the contents of R2 and store the result in R2.

4.9 - INSTRUCTION COUNTER RELATIVE (ICR)

Format: ADDR

Description: This addressing mode is used for 16-bit branch instructions. This mode allows addressing within a memory region of -128 to 127 words relative to the address of the current instruction.

Example:

LABEL	COMMAND	ARGUMENT
	BR	%ALPHA

In the example we must have -128 <= ALPHA-($-1) <= 127, and a branch is made to memory location ALPHA -($-1).

4.10 - SPECIAL ADDRESSING (S)

Format: Not Specific.

Description: This special addressing mode is used where no other addressing mode is applicable.

Example:

LABEL	COMMAND	ARGUMENT
	BRK	#3

This instruction will cause the location counter to be loaded from the third location following the SW in the new processor state.

5 - MIL-STD 1750A DUAL MNEMONICS

This following pages contain each 1750A instruction, its assembly syntax, its addressing modes, the equivalent Mil-Std mnemonic, and a description of the instruction's function. The following terms are utilized in the description:

ADDR = Address
AS = Address State
BR = Base Register (R12, R13, R14 or R15)
BR' = BR-12
CS = Condition Status
DATA = A 16 bit value, address or external
DO = Derived Operand
DSPL = Displacement
IC = Instruction Location Counter
LSH = Least Significant Half
MSH = Most Significant Half
N = A value between 0 and 15 (or 1 and 16)
RD = Destination Register
RN = Register R0...R15 where N is the register number
RS = Source Register
RX = Index Register
SW = Status Word
$ = Current location counter

ABS

Single precision absolute value of register.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format (8 / 4 / 4)
R	ABS	ABS RS,RD	A4 | RD | RS

Description

If the sign bit of the source register, RS, is a one, its negative or 2's complement is stored into register RD. However, if the sign bit of RS is zero, it is stored, unchanged, into RD. The condition status, CS, is set based upon the result in register RD. The absolute value of a number with a 1 in the sign bit and all other bits zero is the same word, and causes fixed point overflow to occur.

ABSF

Floating point absolute value of register.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	FABS	ABSF RS,RD	AC (8) | RD (4) | RS (4)

Description:

If the sign bit of the mantissa of the Derived Operand (DO) is a ones, its floating point negative is stored in registers RD and RD+1. The negative of DO is computed by taking the 2's complement of the mantissa and leaving the exponent unchanged. Exceptions to this are negative powers of two. In other words, the DO mantissa is shifted logically right one position and the exponent incremented. A floating point overflow shall occur if DO is the smallest negative number. If the sign bit of DO is a zero, it is stored unchanged into RD and RD+1. The condition status, CS, is set based on the result in register RD and RD+1. DO is assumed to be a normalized number or floating point zero.

ABSL

Double precision absolute value of register.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
			8 / 4 / 4
R	DABS	ABSL RS,RD	A5 | RD | RS

Description:

If the sign bit of the double precision Derived Operand is a one, its negative or 2's complement is stored into register RD and RD+1, such that register RD contains the MSH of the result. However, if the sign bit of DO is zero, it is stored, unchanged, into RD and RD+1. The condition status, CS, is set based on the result in register RD and RD+1. The absolute value of a number with a 1 in the sign bit and all other bits zero is the same word, and causes fixed point overflow to occur.

ADD

Single precision integer add.

```
 Addr     Mil          Assembler Language          Instruction Format
 Mode     Std               Syntax
                                                       8       4     4
                                                  -------------------------
  R       AR          ADD     RS,RD               |  A1     |  RD |  RS  |
                                                  |_________|_____|______|
                                                     4    2    2     8
                                                  -------------------------
  B       AB          ADD     DSPL(BR),R2         |  1   | 0| BR'|DSPL|
                                                  |______|__|____|____|

                                                    4    2     2     4     4
                                                  ------------------------------
  BX      ABX         ADD     (BR,RX),R2          | 4 | 0 | BR' | 4 | RX |
                                                  |___|___|_____|___|____|

                                                       8       4     4
                                                  -------------------------
  D       A           ADD     ADDR,RD             |  A0     |  RD |  RX  |
                                                  |_________|_____|______|
  DX      A           ADD     ADDR(RX),RD         |        ADDR          |
                                                  |______________________|
                                                       8     4      4
                                                  -------------------------
  IM      AISP        ADD     #DATA,RD            |  A2  | RD  |DATA-1  |
                                                  |______|_____|________|
                                                       1<=DATA<=16

                                                       8       4     4
                                                  -------------------------
                                                  |  4A     |  RD |  1   |
  IM      AIM         ADD     ##DATA,RD           |_________|_____|______|
                                                  |        DATA          |
                                                  |______________________|
                                                       8       4     4
                                                    -----------------------
  D       INCM        ADD     #DATA,ADDR            |   A3  |DATA-1|RX  |
  DX                  ADD     #DATA,ADDR(RX)        |_______|______|____|
                                                    |       ADDR        |
                                                    |___________________|
                                                       1<=DATA<=16
```

Description:
The Source Operand is added to the contents of the destination operand. The result is stored in the destination operand. The condition status (CS) is set based on the result of the destination operand and carry. A fixed point overflow occurs if both operands are of the same sign and the sum is of opposite sign.

ADDF

Floating point add.

```
Addr    Mil     Assembler Language        Instruction Format
Mode    Std          Syntax
                                              8      4    4
                                          -------------------
 R      FAR     ADDF   RS,RD              |  A9   |  RD |  RS |
                                          |_______|_____|_____|

                                             4    2   2    8
                                          -------------------
 B      FAB     ADDF   DSPL(BR),RO        |  2  | 0| BR'|DSPL|
                                          |_____|__|____|____|

                                             4    2   2   4  4
                                          -------------------
 BX     FABX    ADDF   (BR,RX),RO         |  4  | 0|BR'|8 |RX|
                                          |_____|__|___|__|__|

                                              8      4    4
                                          -------------------
 D      FA      ADDF   ADDR,RD            |  A8   | RD  |  RX |
                                          |_______|_____|_____|
 DX     FA      ADDF   ADDR(RX),RD        |       ADDR        |
                                          |___________________|
```

Description:

The floating point source operand is floating point added to the contents of destination registers RD and RD+1. The process of this operation is as follows: the mantissa of the number with the smaller algebraic exponent is shifted right and the exponent incremented by one for each bit shifted until the exponents are equal. The mantissas are then added. If the sum overflows the 24-bit mantissa, then the sum is shifted right one position, the sign bit restored, and the exponent incremented by one. If the exponent exceeds HEX (7F) as a result of this incrementation, overflow occurs and the operation is terminated. If the sum does not result in exponent overflow, the result is normalized. If, in the normalization process, the exponent is decremented below HEX (80), then underflow occurs and a zero is inserted for the result.

ADDL

Double precision integer add.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	DAR	ADDL RS,RD	A7 (8) | RD (4) | RS (4)
D	DA	ADDL ADDR,RD	A6 (8) | RD (4) | RX (4); ADDR
DX	DA	ADDL ADDR(RX),RD	

```
   8      4     4
 ---------------------
 |  A7  |  RD |  RS  |
 |______|_____|______|

   8      4     4
 ---------------------
 |  A6  |  RD |  RX  |
 |______|_____|______|
 |       ADDR        |
 |___________________|
```

Description:

The double precision source operand is added to the contents of the destination operand. The MSH is in the initial location of the destination operand. The condition status (CS) is set based on the double precision results. A fixed point overflow occurs if both operands are of the same sign and the sum is of opposite sign.

ADDX

Extended precision floating point add.

```
Addr    Mil      Assembler Language       Instruction Format
Mode    Std          Syntax
                                            8       4      4
                                          ----------------------
 R      EFAR     ADDX   RS,RD             !  AB   !  RD  !  RS  !
                                          !______!______!______!
                                             8       4      4
                                          ----------------------
 D      EFA      ADDX   ADDR,RD           !  AA   !  RD  !  RX  !
                                          !______!______!______!
 DX     EFA      ADDX   ADDR(RX),RD       !        ADDR         !
                                          !_____________________!
```

<u>Description:</u>

The extended precision floating point source operand is added to the contents of the destination operand. The result is stored in the destination operand. The process of this operation is as follows: the mantissa of the number with the smaller algebraic exponent is shifted right and the exponent is incremented by one for each bit shifted. When the exponents are equal, the mantissas are added. If the sum overflows the 39-bit mantissa, then the sum is shifted right one position, the sign bit restored, and the exponent is incremented by one. If the exponent exceeds HEX(7F) as a result of this incrementation, overflow occurs and the operation is terminated. If the sum does not result in exponent overflow, the result is normalized. If in the normalization process the exponent is decremented below HEX (80), then underflow occurs and a zero is inserted for the result.

AND

Logical and.

```
 Addr      Mil          Assembler Language        Instruction Format
 Mode      Std               Syntax
                                                        8      4    4
                                                   ------------------------
   R       ANDR              AND   RS,RD           |   E3    | RD  | RS  |
                                                   |_________|_____|_____|

                                                      4    2    2    8
                                                   ------------------------
   B       ANDB              AND   DSPL(BR),R2     |  3   | 1| BR' |DSPL|
                                                   |______|__|_____|____|

                                                      4   2    2    4     4
                                                   ------------------------------
   BX      ANDX              AND   (BR,RX),R2      | 4 | 0 | BR'  | E | RX |
                                                   |___|___|______|___|____|

                                                      8       4     4
                                                   ------------------------
   D       AND               AND   ADDR,RD         |  E2  |  RD  |  RX  |
                                                   |______|______|______|
   DX      AND               AND   ADDR(RX),RD     |        ADDR          |
                                                   |______________________|

                                                      8       4     4
                                                   ------------------------
   IM      ANDM              AND   ##DATA,RD       |  4A  |  RD  |   7  |
                                                   |______|______|______|
                                                   |        DATA          |
                                                   |______________________|
```

Description:

The source operand is bit-by-bit logically ANDed with the destination operand. The result is stored in the destination operand. The condition status, CS, is set based on the result in the destination operand.

BC

Branch on carry.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
D	JC,8	BC	ADDR	8 / 4 / 4: 70 | 8 | RX ; ADDR
DX	JC,8	BC	ADDR(RX)	
I	JCI,8	BC	@ADDR	8 / 4 / 4: 71 | 8 | RX ; ADDR
IX	JCI,8	BC	@ADDR(RX)	

```
  8      4     4
-------------------
| 70  |  8  | RX |
|_____|_____|____|
|      ADDR      |
|________________|

  8      4     4
-------------------
| 71  |  8  | RX |
|_____|_____|____|
|      ADDR      |
|________________|
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a carry. Otherwise, the next sequential instruction is executed.

BCGE

Branch if carry or greater than or equal to zero.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
D	JC,E	BCGE	ADDR	8 / 4 / 4: 70 | E | RX ; ADDR
DX	JC,E	BCGE	ADDR(RX)	
I	JCI,E	BCGE	@ADDR	8 / 4 / 4: 71 | E | RX ; ADDR
IX	JCI,E	BCGE	@ADDR(RX)	

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate greater than or equal to zero. Otherwise, the next sequential instruction is executed.

BCLE

Branch if carry or less than or equal to zero.

Addr Mode	Mil Std	Assembler Language Syntax	
D	JC,B	BCLE	ADDR
DX	JC,B	BCLE	ADDR(RX)
I	JCI,B	BCLE	@ADDR
IX	JCI,B	BCLE	@ADDR(RX)

Instruction Format

```
     8        4      4
 ---------------------------
 ¦  70    ¦   B   ¦  RX  ¦
 ¦________¦_______¦______¦
 ¦         ADDR          ¦
 ¦_______________________¦

     8        4      4
 ---------------------------
 ¦  71    ¦   B   ¦  RX  ¦
 ¦________¦_______¦______¦
 ¦         ADDR          ¦
 ¦_______________________¦
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a carry or less than or equal to zero. Otherwise, the next sequential instruction is executed.

BCN

Branch if carry or negative.

Addr Mode	Mil Std	Assembler Language Syntax	
D	JC,9	BCN	ADDR
DX	JC,9	BCN	ADDR(RX)
I	JCI,9	BCN	@ADDR
IX	JCI,9	BCN	@ADDR(RX)

Instruction Format

D, DX:

```
     8        4     4
 ---------------------------
 !   70   !   9   !   RX   !
 !________!_______!________!
 !          ADDR           !
 !_________________________!
```

I, IX:

```
     8        4     4
 ---------------------------
 !   71   !   9   !   RX   !
 !________!_______!________!
 !          ADDR           !
 !_________________________!
```

<u>Description:</u>

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a carry or less than or equal to zero. Otherwise, the next sequential instruction is executed.

BCNZ

Branch if carry or not equal to zero.

Addr Mode	Mil Std	Assembler Language Syntax	
D	JC,D	BCNZ	ADDR
DX	JC,D	BCNZ	ADDR(RX)
I	JCI,D	BCNZ	@ADDR
IX	JCI,D	BCNZ	@ADDR(RX)

Instruction Format

```
      8        4       4
 ------------------------------
 |  70    |   D    |   RX   |
 |________|________|________|
 |           ADDR            |
 |___________________________|

      8        4       4
 ------------------------------
 |  71    |   D    |   RX   |
 |________|________|________|
 |           ADDR            |
 |___________________________|
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a carry or not equal to zero. Otherwise, the next sequential instruction is executed.

BCP

Branch if carry or positive.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
				8 / 4 / 4
D	JC,C	BCP	ADDR	70 / C / RX
DX	JC,C	BCP	ADDR(RX)	ADDR
				8 / 4 / 4
I	JCI,C	BCP	@ADDR	71 / C / RX
IX	JCI,C	BCP	@ADDR(RX)	ADDR

```
 8      4     4
---------------------
| 70   |  C  |  RX |
|______|_____|_____|
|       ADDR       |
|__________________|

 8      4     4
---------------------
| 71   |  C  |  RX |
|______|_____|_____|
|       ADDR       |
|__________________|
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a carry or positive. Otherwise, the next sequential instruction is executed.

BCZ

Branch if carry or equal to zero.

Addr Mode	Mil Std	Assembler Language Syntax	
D	JC,A	BCZ	ADDR
DX	JC,A	BCZ	ADDR(RX)
I	JCI,A	BCZ	@ADDR
IX	JCI,A	BCZ	@ADDR(RX)

Instruction Format

```
       8        4     4
 -------------------------
 |   70    |   A   |  RX |
 |_________|_______|_____|
 |         ADDR          |
 |_______________________|

       8        4     4
 -------------------------
 |   71    |   A   |  RX |
 |_________|_______|_____|
 |         ADDR          |
 |_______________________|
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a carry or equal to zero. Otherwise, the next sequential instruction is executed.

BGE

Branch if greater than or equal to zero.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
ICR	BGE	BGE	%ADDR	8: 7B; 8: ADDR-$ (-128<=ADDR-$<=127)
D	JC,6	BGE	ADDR	8: 70; 4: 6; 4: RX / ADDR
DX	JC,6	BGE	ADDR(RX)	
I	JCI,6	BGE	@ADDR	8: 71; 4: 6; 4: RX / ADDR
IX	JCI,6	BGE	@ADDR(RX)	

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a greater than or equal to zero. Otherwise, the next sequential instruction is executed.

BLE

Branch if less than or equal to zero.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
ICR	BLE	BLE %ADDR	8: 78 | 8: ADDR-$; -128<=ADDR-$<=127
D	JC,3	BLE ADDR	8: 70 | 4: 3 | 4: RX ; ADDR
DX	JC,3	BLE ADDR(RX)	
I	JCI,3	BLE @ADDR	8: 71 | 4: 3 | 4: RX ; ADDR
IX	JCI,3	BLE @ADDR(RX)	

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a less than or equal to zero. Otherwise, the next sequential instruction is executed.

BN

Branch if negative.

```
Addr   Mil         Assembler Language        Instruction Format
Mode   Std              Syntax

                                                 8        8
                                            -------------------
ICR    BLT         BN      %ADDR            |   76    | ADDR-$ |
                                            |_________|________|
                                             -128<=ADDR-$<=127

                                                 8      4     4
                                            -------------------
D      JC,1        BN      ADDR             |  70   |  1  | RX |
                                            |_______|_____|____|
DX     JC,1        BN      ADDR(RX)         |       ADDR       |
                                            |__________________|

                                                 8      4     4
                                            -------------------
I      JCI,1       BN      @ADDR            |  71   |  1  | RX |
                                            |_______|_____|____|
IX     JCI,1       BN      @ADDR(RX)        |       ADDR       |
                                            |__________________|
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address if the condition codes indicate a negative. Otherwise, the next sequential instruction is executed.

BNZ

Branch if not zero.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
ICR	BNZ	BNZ %ADDR	8: 7A; 8: ADDR-$ (-128<=ADDR-$<=127)
D	JC,5	BNZ ADDR	8: 70; 4: 5; 4: RX / ADDR
DX	JC,5	BNZ ADDR(RX)	
I	JCI,5	BNZ @ADDR	8: 71; 4: 5; 4: RX / ADDR
IX	JCI,5	BNZ @ADDR(RX)	

```
      8        8
 -------------------
 |   7A   | ADDR-$ |
 |________|________|
 -128<=ADDR-$<=127

      8      4     4
 -------------------
 |  70  |  5  | RX |
 |______|_____|____|
 |      ADDR       |
 |_________________|

      8      4     4
 -------------------
 |  71  |  5  | RX |
 |______|_____|____|
 |      ADDR       |
 |_________________|
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address, if the condition codes indicate not zero. Otherwise, the next sequential instruction is executed.

BP

Branch if positive.

```
Addr    Mil            Assembler Language         Instruction Format
Mode    Std                  Syntax

                                                     8         8
                                                 ---------------------
ICR     BGT            BP      %ADDR             :   79    :  ADDR-$  :
                                                 :_________:__________:
                                                  -128<=ADDR-$<=127

                                                    8        4     4
                                                 ---------------------
D       JC,4           BP      ADDR              :  70   :  4   :  RX :
                                                 :_______:______:_____:
DX      JC,4           BP      ADDR(RX)          :        ADDR        :
                                                 :____________________:

                                                    8        4     4
                                                 ---------------------
I       JCI,4          BP      @ADDR             :  71   :  4   :  RX :
                                                 :_______:______:_____:
IX      JCI,4          BP      @ADDR(RX)         :        ADDR        :
                                                 :____________________:
```

Description:

This is a conditional jump instruction wherein the instruction sequence jumps to the derived address, if the condition codes indicate a positive. Otherwise, the next sequential instruction is executed.

BR

Branch unconditionally.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
ICR	BR	BR %ADDR	8: 74 | 8: ADDR-$; -128<=ADDR-$<=127
D	JC,F	BR ADDR	8: 70 | 4: F | 4: RX / ADDR
DX	JC,F	BR ADDR(RX)	
I	JCI,F	BR @ADDR	8: 71 | 4: F | 4: RX / ADDR
IX	JCI,F	BR @ADDR(RX)	

Description:

This is an unconditional jump instruction wherein the instruction jumps to the derived address.

BRK

Branch to executive.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
S	BEX	BRK	#N	77	0	N

Description:

This instruction provides a means to jump to a routine in another address state (AS). It is typically used to make controlled, protected calls to an executive. The 4-bit value N selects one of 16 executive entry points to be used. Execution of this instruction causes an interrupt to occur using the EXEC call interrupt vector (interrupt 5). The new $ is loaded from the Nth location following the SW in the next processor state. The linkage pointer (LP), service pointer (SP), and the new processor state (new MK, new SW, and new $) are fetched from address state zero. The current processor state is stored in the address state specified by the new SW AS field. Interrupts are disabled when BRK is executed. The EXEC call interrupt cannot be masked or disabled. Arguments associated with the BRK instruction are passed by software convention. The processor lock and key function is ignored when this instruction is executed. An attempt to branch into an execute protected area of memory shall result in FT(0) being set to 1.

BZ

Branch if equal to zero.

```
 Addr      Mil         Assembler Language          Instruction Format
 Mode      Std               Syntax
                                                        8         8
                                                   ---------------------
  ICR      BEZ            BZ      %ADDR            |   75    |  ADDR-$ |
                                                   |_________|_________|
                                                     -128<=ADDR-$<=127

                                                       8      4      4
                                                   ---------------------
  D        JC,2           BZ      ADDR             |  70   |  2   |  RX |
                                                   |_______|______|_____|
  DX       JC,2           BZ      ADDR(RX)         |        ADDR        |
                                                   |____________________|

                                                       8      4      4
                                                   ---------------------
  I        JCI,2          BZ      @ADDR            |  71   |  2   |  RX |
                                                   |_______|______|_____|
  IX       JCI,2          BZ      @ADDR(RX)        |        ADDR        |
                                                   |____________________|
```

Description:

A program branch is made to the derived address if the condition status, CS, indicates that the previous result which set the CS is equal to zero. Otherwise, the next sequential instruction is executed.

CALL

Jump to subroutine.

Addr Mode	Mil Std	Assembler Language Syntax
D	JS	CALL RS,ADDR
DX	JS	CALL RS,ADDR(RX)
D	SJS	CALL @-RS,ADDR
DX	SJS	CALL @-RS,ADDR(RX)

Instruction Format

```
      8       4     4
 ----------------------
 :  72   :  RS  :  RX  :
 :_______:______:______:
 :        ADDR         :
 :_____________________:

      8       4     4
 ----------------------
 :  7E   :  RS  :  RX  :
 :_______:______:______:
 :        ADDR         :
 :_____________________:
```

Description:

The value of the instruction counter (the address of the next sequential instruction) is stored into the effective source register. Then, the current location counter ($) is set to the derived address, DA, thus effecting the jump. This sets up the return from the subroutine to the address stored in the register.

CLRI

Clear static bit.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
			8 / 4 / 4
R	RBR	CLRI #N,RD	54 | N | RD
			8 / 4 / 4
D	RB	CLRI #N,ADDR	53 | N | RX
DX	RB	CLRI #N,ADDR(RX)	ADDR
			8 / 4 / 4
I	RBI	CLRI #N,@ADDR	55 | N | RX
IX	RSI	CLRI #N,@ADDR(RX)	ADDR
			8 / 4 / 4
R	RVBR	CLRI RS,RD	5C | RD | RS

Description:

Bit number N (0<=N<=15) of the destination operand is set to zero. If a register is used instead of N, then the least significant four bits determine which bit.

CMP

Single precision compare.

```
 Addr      Mil      Assembler Language         Instruction Format
 Mode      Std           Syntax
                                                  8       4     4
                                              -------------------
   R       CR        CMP    RS,RD             |  F1   | RD  | RS  |
                                              |_______|_____|_____|
                                                 4   2    2     8
                                              -------------------
   B       CB        CMP    R2,DSPL(BR)       |  3  | 2| BR'|DSPL|
                                              |_____|__|____|____|

                                                 4  2    2     4     4
                                              -------------------------
   BX      CBX       CMP    R2,(BR,RX)        | 4 | 0 | BR' | C | RX |
                                              |___|___|_____|___|____|

                                                  8       4     4
                                              -------------------
   D       C         CMP    RS,ADDR           |  F0   |  RS |  RX |
                                              |_______|_____|_____|
   DX      C         CMP    RS,ADDR(RX)       |       ADDR        |
                                              |___________________|
                                                  8     4     4
                                              -------------------
   IM      CISP      CMP    RS,#DATA          |  F2 | RS |DATA-1 |
                                              |_____|____|_______|
                                                 1<=DATA<=16

                                                  8     4     4
                                              -------------------
   IM      CISN      CMP    RS,#DATA          |  F3 | RS | DATA-1|
                                              |_____|____|_______|
                                                  16<=DATA<=1

                                                  8       4     4
                                              -------------------
                                              |  4A   |  RS |  A  |
   IM      CIM       CMP    RS,##DATA         |_______|_____|_____|
                                              |       DATA        |
                                              |___________________|
```

Description: The single precision source operand is compared to the contents of the destination operand. Then the Condition Status, CS, is set based on whether the contents of the destination operand is less than, equal to, or greater than the destination operand.

CMPF

Floating point compare.

```
Addr        Mil      Assembler Language        Instruction Format
Mode        Std          Syntax
                                                 8       4     4
                                              -------------------
 R          FCR       CMPF   RS,RD            !  F9   ! RD  !  RS !
                                              !______!_____!_____!

                                                4    2    2    8
                                              -------------------
 B          FCB       CMPF   RO,DSPL(BR)      !  3  ! 3! BR'!DSPL!
                                              !_____!__!____!____!

                                                4  2    2    4    4
                                              ---------------------
 BX         FCBX      CMPF   RO,(BR,RX)       ! 4 ! O ! BR' ! D ! RX !
                                              !___!___!_____!___!____!

                                                 8       4     4
                                              -------------------
 D          FC        CMPF   RS,ADDR          !  F8  !  RS ! RX  !
                                              !______!_____!_____!
 DX         FC        CMPF   RS,ADDR(RX)      !      ADDR        !
                                              !__________________!
```

Description:

The floating point number in the destination operand is compared to the floating point Derived Operand. Then, the Condition Status, CS, is set based on whether the contents are less than, equal to, or greater than the Derived Operand. The contents of the operands are unchanged.

CMPL

Double precision compare.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	DCR	CMPL RS,RD	8: F7 / 4: RD / 4: RS
D	DC	CMPL RS,ADDR	8: F6 / 4: RS / 4: RX ; ADDR
DX	DC	CMPL RS,ADDR(RX)	

```
 8        4     4
----------------------
| F7    | RD   | RS  |
|_______|______|_____|

 8        4     4
----------------------
| F6    | RS   | RX  |
|_______|______|_____|
|       ADDR         |
|____________________|
```

Description:

The double precision derived operand is compared to the contents of the source registers RS and RS+1 where RS contains the MSH of a double precision word. Then, the Condition Status, CS, is set based on whether the contents of RS,RS+1 is less than, equal to, or greater than the derived operand. The contents of RS and RS+1 are unchanged.

CMPRNG

Compare between limit.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
D	CBL	CMPRNG	RS,ADDR	
DX	CBL	CMPRNG	RS,ADDR(RX)	

```
     8        4      4
 ----------------------------
 |   F4   |  RS   |   RX   |
 |________|_______|________|
 |          ADDR           |
 |_________________________|
```

Description:

The contents of register RS are compared to two different sixteen bit derived operands, DO1 and DO2. The derived operands, DO1 and DO2 are located at DA and DA+1, respectively, and their values are defined such that DO1<=DO2. The CS is set based on the result. If the values for DO1 and DO2 are defined incorrectly (that is, DO1>DO2), then CS is set to 1000.

CMPX

Extended precision floating point compare.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	EFCR	CMPX	RS,RD	FB (8) | RD (4) | RS (4)
D	EFC	CMPX	RS,ADDR	FA (8) | RS (4) | RX (4) / ADDR
DX	EFC	CMPX	RS,ADDR(RX)	

Description:

The extended precision floating derived operand is compared to the contents of registers RS, RS+1, and RS+2 where RS contains the most significant 16-bits of the extended precision floating point word. The condition status, CS, is set based on whether the contents of RS,RS+1, and RS+2 are less than, equal to, or greater than the DO. The contents of RS, RS+1, and RS+2 are unchanged.

DECBNZ

Decrement one and branch if non-zero.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
D	SOJ	DECBNZ	RS,ADDR	73 (8) | RS (4) | RX (4)
DX	SOJ	DECBNZ	RS,ADDR(RX)	ADDR

Description:

The 16 bit contents of register RS are decremented by one. Then if the content of register RS is zero, the next sequential instruction is executed. If the contents of register RS is non-zero, then a jump to the derived address occurs.

DIV

Single precision integer divide with 16-bit dividend.

```
 Addr      Mil          Assembler Language          Instruction Format
 Mode      Std               Syntax

                                                      8     4     4
                                                   ------------------
  R        DVR          DIV    RS,RD               |  D1 | RD |  RS  |
                                                   |_____|____|______|

                                                      8       4     4
                                                   -------------------
  D        DV           DIV    ADDR,RD             |  DO  | RD  | RX |
                                                   |______|_____|____|
  DX       DV           DIV    ADDR(RX),RD         |       ADDR       |
                                                   |__________________|

                                                      8     4     4
                                                   -------------------
  IM       DISP         DIV    #DATA,RD            |  D2 | RD |DATA-1 |
                                                   |_____|____|_______|
                                                       1<=DATA<=16
                                                      8     4     4
                                                   -------------------
  IM       DISN         DIV    #DATA,RD            |  D3 | RD | DATA-1|
                                                   |_____|____|_______|
                                                        16<=DATA<=1
                                                      8       4     4
                                                   -------------------
                                                   |  4A  |  RD |  6 |
  IM       DVIM         DIV    ##DATA,RD           |______|_____|____|
                                                   |       DATA       |
                                                   |__________________|
```

Description:

The contents of register RD are divided by the source operand, a single precision, 2's complement number. The result is stored in registers RD and RD+1 such that RD stores the single precision integer quotient and RD+1 stores the remainder. The Condition Status, CS, is set based on the result in RD. A fixed point overflow occurs if the divisor is zero, or if the dividend is HEX(8000) and the divisor is HEX(FFFF).

DIVF

Floating point divide.

```
Addr     Mil       Assembler Language          Instruction Format
Mode     Std            Syntax

                                                  8     4     4
                                               ------------------
  R      FDR         DIVF  RS,RD               |  D9 | RD |  RS  |
                                               |_____|____|______|

                                                 4    2    2    8
                                               --------------------
  B      FDB         DIVF  DSPL(BR),RO         |  2  | 3| BR'|DSPL|
                                               |_____|__|____|____|

                                                 4  2     2    4    4
                                               -------------------------
  BX     FDBX        DIVF  (BR,RX),RO          | 4 | 0 | BR'  | B | RX |
                                               |___|___|______|___|____|

                                                  8      4      4
                                               ---------------------
  D      FD          DIVF  ADDR,RD             |  D8   | RD  |  RX |
                                               |_______|_____|_____|
  DX     FD          DIVF  ADDR(RX),RD         |       ADDR        |
                                               |___________________|
```

Description:

The floating point number in registers RD and RD+1 is divided by the floating point Derived Operand. The result is stored in register RD and RD+1. A floating point overflow occurs if the exponent result exceeds HEX(7F) at any point in the calculation process. Underflow occurs if the exponent result is less than HEX(80) at any point in the process. If underflow occurs, then the quotient is forced to zero. A divide by zero yields a floating point overflow.

DIVL

Double precision integer divide.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format 8	4	4
R	DDR	DIVL	RS,RD	D7	RD	RS
D	DD	DIVL	ADDR,RD	D6	RD	RX
DX	DD	DIVL	ADDR(RX),RD	ADDR		

Description:

The contents of registers RD and RD+1, a double precision 2's complement number, are divided by the source operand, a double precision 2's complement number. RD contains the MSH of the 32-bit dividend. The quotient part of the integer result is stored in registers RD and RD+1 (with the MSH in RD) and the remainder is lost. The Condition Status, CS, is set based on the results in registers RD and RD+1. A fixed point overflow occurs if the divisor is zero, or if the dividend is HEX(8000) and the divisor is HEX(FFFF).

DIVQ

Single precision integer divide with 32-bit dividend.

```
Addr   Mil     Assembler Language       Instruction Format
Mode   Std          Syntax
                                          8     4     4
                                        ------------------
 R     DR        DIVQ  RS,RD            |  D5  | RD |  RS  |
                                        |______|____|______|
                                           4    2   2    8
                                        -------------------
 B     DB        DIVQ  DSPL(BR),R2      |  1  | 3| BR'|DSPL|
                                        |_____|__|____|____|
                                           4  2    2    4     4
                                        -----------------------
 BX    DBX       DIVQ  (BR,RX),R2       | 4 | 0 | BR' | 7 | RX |
                                        |___|___|_____|___|____|

                                          8       4     4
                                        --------------------
 D     D         DIVQ  ADDR,RD          |  D4  |  RD  |  RX |
                                        |______|______|_____|
 DX    D         DIVQ  ADDR(RX),RD      |       ADDR        |
                                        |___________________|
                                          8       4     4
                                        --------------------
 IM    DIM       DIVQ  ##DATA,RD        |  4A  |  RD |  5  |
                                        |______|_____|_____|
                                        |       DATA        |
                                        |___________________|
```

Description:

The source operand is multiplied by the contents of the destination registers. The result is stored in register RD and RD+1. The process of the operation is as follows: the exponents of the operands are added. If the sum exceeds HEX(7F), a floating point overflow occurs. If the sum is less than HEX(80), then underflow occurs and the result is set to zero. The operand mantissas are multiplied and the result normalized and stored in RD and RD+1. It is possible that the normalization process may yield an exponent underflow; if this occurs, then the result is forced to zero. The condition status, CS, is set based on the result in RD and RD+1.

DIVX

Extended precision floating point divide.

```
 Addr     Mil      Assembler Language          Instruction Format
 Mode     Std           Syntax

                                                   8      4     4
                                               -------------------
  R       EFDR      DIVX      RS,RD            |  DB  |  RD  |  RS |
                                               |______|______|_____|

                                                   8      4     4
                                               -------------------
  D       EFD       DIVX      ADDR,RD          |  DA  |  RD  |  RX |
                                               |______|______|_____|
  DX      EFD       DIVX      ADDR(RX),RD      |       ADDR        |
                                               |___________________|
```

<u>Description:</u>

The contents of registers RD, RD+1 and RD+2 are extended precision floating point divided by the extended precision floating point Derived Operand. The result is stored in the destination operand. A floating point overflow occurs if the exponent result exceeds HEX(7F) at any point in the calculation process. Underflow occurs if the exponent result is less than HEX(80) at any point if the process. If underflow occurs, then the quotient is forced to zero. A divide by zero yields a floating point overflow.

FLOAT

Convert floating point to 16-bit integer.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	FLT	FLOAT	RS,RD	E9	RD	RS

Description:

The integer Derived Operand, DO (i.e., the contents of register RS), is converted to Single Precision floating point format and stored in register RD and RD+1. The condition status, CS, is set based on the results in RD and RD+1. The operation process is as follows: The exponent is initially considered to be HEX(7F). The integer value in RS is normalized, i.e., the number is left shifted and the exponent decremented for each shift until the sign bit and the next MSB are unequal, and the exponent and mantissa stored in the proper fields of RD and RD+1.

FLOATL

Convert 32-bit integer to extended precision floating point.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	EFLT	FLOATL RS,RD	EB (8) | RD (4) | RS (4)

Description:

The double precision integer Derived Operand (i.e., the contents of registers RS and RS+1), is converted to Extended Precision Floating Point format and stored in the destination operand. The condition status, CS, is set based on the result in the destination operand. The operation process is as follows: The exponent is initially considered to be HEX(1F). The integer value in RS, RS+1 is normalized, i.e., the number is left shifted and the exponent decremented for each shift until the sign bit and the next MSB are unequal, and the exponent and mantissa stored in the proper field of RD, RD+1 and RD+2.

HALT

Break point.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format		
			8	4	4
S	BPT	HALT	FF	F	F

<u>Description:</u>

This instruction is typically used for halting the processor during maintenance and diagnostic procedures when the maintenance console is connected to the system. If the console is not connected, this instruction is treated as a NOP. Restarting the processor after a BPT can only be done by the maintenance console or the power on sequence.

INTGR

Convert floating point to 16-bit integer.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
			8 / 4 / 4
R	FIX	INTGR RS,RD	E8 | RD | RS

Description:

The integer portion of the floating point Derived Operand (i.e., the contents of registers RS and RS+1), is stored in register RD. If the actual value of the DO floating point exponent is greater than HEX (OF), then RD remains unchanged and a fixed point overflow occurs. The condition status, CS, is set based on the result in RD. The algorithm truncates toward zero.

INTGRX

Convert extended precision floating point to 32-bit integer.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	EFIX	INTGRX RS,RD	8 / 4 / 4

```
  8       4     4
 ----------------------
 |  EA  |  RD  |  RS  |
 |______|______|______|
```

Description:

The integer portion of the floating point Derived Operand (i.e., the contents of registers RS, RS+1 and RS+2), is stored into register RD and RD+1. If the actual value of the DO floating point exponent is greater than HEX (1F), then RD and RD+1 remain unchanged and a fixed point overflow occurs. The condition status, CS, is set based on the result in RD and RD+1. The algorithm truncates toward zero.

LD

Single precision load.

```
Addr     Mil        Assembler Language        Instruction Format
Mode     Std             Syntax
                                                 8      4     4
                                              ---------------------
 R       LR        LD     RS,RD               |  81   | RD  | RS  |
                                              |_______|_____|_____|
                                                4   2    2    8
                                              ---------------------
 B       LB        LD     DSPL(BR),R2         |  0   | 0| BR'|DSPL|
                                              |______|__|____|____|

                                               4    2    2    4    4
                                              -------------------------
 BX      LBX       LD     (BR,RX),R2          | 4 | 0 | BR' | 0 | RX |
                                              |___|___|_____|___|____|

                                                 8     4     4
                                              ---------------------
 IM      LISP      LD     #DATA,RD            |  82  | RD | DATA-1|
                                              |______|____|_______|
                                                 1<=DATA<=16
                                              ---------------------
 IM      LISN      LD     #DATA,RD            |  83  | RD | DATA-1|
                                              |______|____|_______|
                                                 16<=DATA<=1
                                                 8      4     4
                                              ---------------------
                                              |  85   | RD  | RX  |
 IM      LIM       LD     ##DATA,RD           |_______|_____|_____|
                   LD     ##DATA(RX),RD       |       DATA        |
                                              |___________________|
                                                 8      4     4
                                              ---------------------
 D       L         LD     ADDR,RD             |  80   | RD  | RX  |
                                              |_______|_____|_____|
 DX      L         LD     ADDR(RX),RD         |       ADDR        |
                                              |___________________|
                                                 8      4     4
                                              ---------------------
 I       LI        LD     @ADDR,RD            |  84   | RD  | RX  |
                                              |_______|_____|_____|
 IX      LI        LD     @ADDR(RX),RD        |       ADDR        |
                                              |___________________|
```

Description:

The single precision source operand is loaded into the destination register. The Condition Status is set based on the result in RD.

LDB

Load byte.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
D	LLB/LUB	LDB	ADDR,RbD	8C/8B (8) | RD (4) | RX (4)
DX	LLB/LUB	LDB	ADDR(RX),RbD	ADDR
I	LLBI/LUBI	LDB	@ADDR,RbD	8E/8D (8) | RD (4) | RX (4)
IX	LLBI/LUBI	LDB	@ADDR(RX),RbD	ADDR

Description:

The designated byte of the derived operand is loaded into the lower byte of register RD. The upper byte of RD is unaffected. The condition status, CS, is set based on the result in RD.

LDL

Double precision load.

```
Addr      Mil        Assembler Language       Instruction Format
Mode      Std             Syntax

                                                   8      4     4
                                               ---------------------
  R       DLR        LDL    RS,RD              |  87   |  RD  | RS  |
                                               |_______|______|_____|

                                                  4    2    2     8
                                               ---------------------
  B       DLB        LDL    DSPL(BR),RO        |  0   | 1| BR' |DSPL|
                                               |______|__|_____|____|

                                                 4    2     2     4    4
                                               -------------------------
  BX      DLBX       LDL    (BR,RX),RO         | 4 | 0 | BR'  | 1 | RX |
                                               |___|___|______|___|____|

                                                   8      4     4
                                               ---------------------
  D       DL         LDL    ADDR,RD            |  86   |  RD  | RX  |
                                               |_______|______|_____|
  DX      DL         LDL    ADDR(RX),RD        |        ADDR        |
                                               |____________________|

                                                  8      4     4
                                               ---------------------
  I       DLI        LDL    @ADDR,RD           |  88   |  RD  | RX  |
                                               |_______|______|_____|
  IX      DLI        LDL    @ADDR(RX),RD       |        ADDR        |
                                               |____________________|
```

Description:

The double precision Derived Operand is loaded into register RD and RD+1 such that the MSH of DO is in RD. The CS is set based on the result in RD and RD+1.

LDM

Load multiple registers.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
D	LM	LDM ADDR,R0,#N	8: 89 | 4: N | 4: RX
DX	LM	LDM ADDR(RX),R0,#N	ADDR

Description:

The contents of the Derived Address are loaded into Register R0, then the contents of the DA+1 are loaded into register R1,....finally, the contents of DA+N are loaded into RN. Effectively, this allows the transfer of (N+1) words from memory to the register file.

LDST

Load status.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
D	LST	LDST ADDR	8 / 4 / 4: 7D | 0 | RX ; ADDR
DX	LST	LDST ADDR(RX)	
I	LSTI	LDST @ADDR	8 / 4 / 4: 7C | 0 | RX ; ADDR
IX	LSTI	LDST @ADDR(RX)	

Description:

The contents of the Derived Address, DA, DA+1, and DA+2 are loaded into the Interrupt Mask register, Status Word register and Instruction Counter, respectively. This is a privileged instruction. This instruction is an unconditional jump and is typically used to exit from an interrupt routine. DA, DA+1, and DA+2, in this typical case, contain the Interrupt Mask, Status Word, and Instruction Counter values for the interrupted program and the execution of LDST causes the program to return to its status prior to being interrupted.

LDX

Extended precision floating point load.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
D	EFL	LDX ADDR,RD	8: 8A | 4: RD | 4: RX
DX	EFL	LDX ADDR(RX),RD	ADDR

Description:

The extended precision floating point Derived Operand is loaded into register RD, RD+1, and RD+2. The condition status, CS, is set based on the results in registers RD, RD+1, and RD+2.

MOVBK

Move multiple words, memory-to-memory.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format		
			8	4	4
S	MOV	MOVBK @RS,@RD	93	RD	RS

Description:

This instruction allows the memory-to-memory transfer of N words, where N is an integer between zero and (2 to the 16th power -1) and is represented by the contents of RD+1. The contents of RS are the address of the first word to be transferred and the contents of RD are the address of where the first word is to be transferred. After each word is transferred, RD and RS are incremented, and RD+1 is decremented. Any pending interrupts are honored after each single word transfer is completed. The location counter points to the current instruction location until the last transfer is completed. RD has a final value of the last stored address plus one; RD+1 has a final value of zero. RS has a final value equal to the address of the last word transferred plus one.

MUL

Single precision integer multiply with 16-bit product.

```
Addr     Mil       Assembler Language        Instruction Format
Mode     Std            Syntax

                                                 8      4     4
                                              -------------------
 R       MSR        MUL    RS,RD              |  C1  | RD  | RS  |
                                              |______|_____|_____|

                                                 8      4     4
                                              -------------------
 D       MS         MUL    ADDR,RD            |  CO  |  RD |  RX |
                                              |______|_____|_____|
 DX      MS         MUL    ADDR(RX),RD        |       ADDR       |
                                              |__________________|

                                                 8    4     4
                                              -------------------
 IM      MSIP       MUL    #DATA,RD           |  C2 | RD | DATA-1|
                                              |_____|____|_______|
                                                1<=DATA<=16

                                                 8    4     4
                                              -------------------
 IM      MSIN       MUL    #DATA,RD           |  C3 | RD | DATA-1|
                                              |_____|____|_______|
                                                16<=DATA<=1

                                                 8      4     4
                                              -------------------
 IM      MSIM       MUL    ##DATA,RD          |  4A  |  RD |  4  |
                                              |______|_____|_____|
                                              |       DATA       |
                                              |__________________|
```

Description:

The derived operand is multiplied by the contents of register RD. The LSH of the result, a 16-bit, 2's complement integer, is stored in register RD. The CS is set based on the result in register RD. A fixed point overflow occurs if (1) both operands are of the same sign and the MSH of the product is not zero, or (2) if the operands are of opposite sign and the MSH of the product is not HEX(FFFF), or the sign bit of the LSH is not 1. A fixed point overflow does not occur if either of the operands is zero.

MULF

Floating point multiply.

```
 Addr      Mil        Assembler Language          Instruction Format
 Mode      Std              Syntax

                                                       8      4     4
                                                   -------------------
   R       FMR         MULF    RS,RD               |  C9  |  RD |  RS |
                                                   |______|_____|_____|

                                                     4    2    2    8
                                                   -------------------
   B       FMB         MULF    DSPL(BR),RO         |  2 |   2|BR' |DSPL|
                                                   |____|____|____|____|

                                                   4    2    2    4    4
                                                   ------------------------
   BX      FMBX        MULF    (BR,RX),RO          | 4 | 0 | BR' | A | RX |
                                                   |___|___|_____|___|____|

                                                       8      4     4
                                                   -------------------
   D       FM          MULF    ADDR,RD             |  C8  |  RD |  RX |
                                                   |______|_____|_____|
   DX      FM          MULF    ADDR(RX),RD         |       ADDR        |
                                                   |___________________|
```

Description:

The floating point Derived Operand is floating point multiplied by the contents of register RD and RD+1. The result is stored in register RD and RD+1. The process of this operation is as follows: the exponents of the operands are added. If the sum exceeds HEX(7F), a floating point overflow occurs. If the sum is less than HEX(80), then underflow occurs and the result is set to zero. The operand mantissas are multiplied and the result normalized and stored in RD and RD+1. An exceptional case is when both operands are negative powers of two. It is possible that the normalization process may yield an exponent underflow; if this occurs, then the result is forced to zero. The CS is set based on the result in RD and RD+1.

MULL

Double precision integer multiply.

Addr Mode	Mil Std	Assembler Language Syntax	
R	DMR	MULL	RS,RD
D	DM	MULL	ADDR,RD
DX	DM	MULL	ADDR(RX),RD

Instruction Format

```
    8       4     4
 -------------------
 |  C7   | RD  | RS  |
 |_______|_____|_____|

    8       4     4
 -------------------
 |  C6   | RD  | RX  |
 |_______|_____|_____|
 |       ADDR        |
 |___________________|
```

Description:

The double precision Derived Operand, a 32-bit 2's complement number, is multiplied by the contents of registers RD and RD+1, a 32-bit 2's complement number, with the MSH in RD. The LSH of the product is retained in RD and RD+1 as a 32-bit, 2's complement number. The MSH is lost. The Condition Status, CS, is set based on the double precision result in registers RD and RD+1. A fixed point overflow occurs if (1) both operands are of the same sign and the MSH of the product is not zero, or the sign bit of the LSH is not zero, or (2) if the operands are of opposite sign and the MSH of the product is not HEX(FFFF), or the sign bit of the LSH is not 1. A fixed point overflow does not occur if either of the operands is zero.

MULQ

Single precision integer multiply with 32-bit product.

```
 Addr     Mil          Assembler Language         Instruction Format
 Mode     Std               Syntax

                                                     8       4      4
                                                 ----------------------
  R       MR            MULQ    RS,RD            |  C5    | RD   | RS   |
                                                 |_______|______|______|

                                                   4    2    2     8
                                                 ----------------------
  B       MB            MULQ    DSPL(BR),R2      |  1    | 2| BR' |DSPL|
                                                 |______|__|_____|____|

                                                  4    2     2     4    4
                                                 ---------------------------
  BX      MBX           MULQ    (BR,RX),R2       | 4 | 0 | BR'  | 6 | RX |
                                                 |___|___|______|___|____|

                                                     8       4      4
                                                 ----------------------
  D       M             MULQ    ADDR,RD          |  C4   |  RD  |  RX  |
                                                 |_______|______|______|
  DX      M             MULQ    ADDR(RX),RD      |        ADDR          |
                                                 |_____________________|

                                                    8       4     4
                                                 ----------------------
  IM      MIM           MULQ    ##DATA,RD        |  4A   |  RD  |  3   |
                                                 |_______|______|______|
                                                 |        DATA          |
                                                 |_____________________|
```

Description:

The Derived Operand is multiplied by the contents of register RD. The result, a 32-bit, 2's complement integer, is stored in registers RD and RD+1 with the MSH of the product in register RD. The Condition Status, CS, is set based on the result in registers RD and RD+1. A special case occurs when DO = (RD) = HEX (8000) (the largest negative number). In this case, DO X (RD) = HEX(4000 0000).

MULX

Extended precision floating point multiply.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	EFMR	MULX	RS,RD	8 / 4 / 4: CB | RD | RS
D	EFM	MULX	ADDR,RD	8 / 4 / 4: CA | RD | RX ; ADDR
DX	EFM	MULX	ADDR(RX),RD	

Description: The extended precision floating point Derived Operand is extended floating point multiplied by the contents of registers RD, RD+1, and RD+2. The result is stored in registers RD, RD+1, and RD+2. The process of the operation is as follows: the exponent of the operands are added. If the sum exceeds HEX(7F), a floating point overflow occurs. If the sum is less than HEX(80), then underflow occurs and the result is set to zero. The operand's mantissas are multiplied and the result normalized and stored in RD, RD+1, and RD+2. The condition status, CS, is set based on the result in RD, RD+1 and RD+2.

NAND

Logical nand.

```
 Addr     Mil        Assembler Language          Instruction Format
 Mode     Std             Syntax

                                                   8        4     4
                                                ------------------------
   R      NR         NAND     RS,RD             ¦  E7   ¦  RD  ¦  RS  ¦
                                                ¦_______¦______¦______¦

                                                   8        4     4
                                                ------------------------
   D      N          NAND     ADDR,RD           ¦  E6   ¦  RD  ¦  RX  ¦
                                                ¦_______¦______¦______¦
   DX     N          NAND     ADDR(RX),RD       ¦        ADDR         ¦
                                                ¦_____________________¦

                                                   8        4     4
                                                ------------------------
   IM     NIM        NAND     ##DATA,RD         ¦  4A   ¦  RD  ¦  B   ¦
                                                ¦_______¦______¦______¦
                                                ¦        DATA         ¦
                                                ¦_____________________¦
```

Description:

The Derived Operand is bit-by-bit logically NANDed with the contents of register RD. The result is stored in RD.

NEG

Single precision negate register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	NEG	NEG	RS,RD	B4	RD	RS

Description:

The negative (i.e., the 2's complement) of the RS register is stored into register RD. The condition status, CS, is set based on the result in register RD. The negative of zero is zero. The negative of a number with a 1 in the sign bit and all other bits zero is the same word, and causes fixed point overflow to occur.

NEGF

Floating point negate register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	FNEG	NEGF	RS,RD	BC (8) | RD (4) | RS (4)

Description:

The 24-bit mantissa of the floating point number in registers RS and RS+1, is 2's complemented. The exponent remains unchanged. The result, the negative of the original number, is stored in RD and RD+1. The 2's complement of a floating point zero is a floating point zero. Exceptions to this are all the powers of two, when the mantissa is either HEX(8000 00) or HEX(4000 00). A floating point overflow occurs for the negation of the smallest negative number. A floating point underflow occurs for the negation of the smallest positive number and causes the result to be zero. The condition status, CS, is set based on the result in registers RD and RD+1.

NEGL

Double precision negate register.

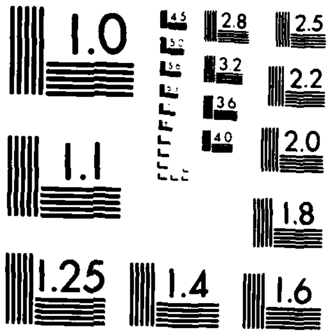
Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	DNEG	NEGL	RS,RD	B5	RD	RS

Description:

The negative (i.e., the 2's complement) of the contents of register RS and RS+1 is stored into registers RD and RD+1 such that register RD contains the MSH of the result. The condition status, CS, is set based on the result in register RD and RD+1. The negative of zero is zero. The negative of a number with a 1 in the sign bit and all other bits zero is the same word, and causes fixed point overflow to occur.

AD-A150 584 PROCEEDINGS OF THE TECHNICAL FORUM (3RD) ON THE F-16 MIL-STD-1750A MICROP..(U) AERONAUTICAL SYSTEMS DIV WRIGHT-PATTERSON AFB OH J L PESLER ET AL. 06 MAY 82 4/6

UNCLASSIFIED ASD-TR-82-5011-VOL-2 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

NOP

No operation.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
S	NOP	NOP	8: FF / 4: 0 / 4: 0

Description:

No operation is performed.

OR

Inclusive logical or.

```
 Addr      Mil        Assembler Language         Instruction Format
 Mode      Std              Syntax

                                                     8        4     4
                                                 ------------------------
   R        ORR          OR    RS,RD             ¦  E1    ¦ RD   ¦ RS   ¦
                                                 ¦________¦______¦______¦

                                                    4    2     2      8
                                                 ------------------------
   B        ORB          OR    DSPL(BR),R2       ¦  3    ¦ 0¦ BR´ ¦DSPL ¦
                                                 ¦_______¦__¦_____¦_____¦

                                                   4    2     2      4     4
                                                 -----------------------------
   BX       ORBX         OR    (BR,RX),R2        ¦ 4 ¦ 0 ¦ BR´  ¦ F  ¦ RX  ¦
                                                 ¦___¦___¦______¦____¦_____¦

                                                     8        4     4
                                                 ------------------------
   D        OR           OR    ADDR,RD           ¦  E0    ¦ RD   ¦ RX   ¦
                                                 ¦________¦______¦______¦
   DX       OR           OR    ADDR(RX),RD       ¦        ADDR          ¦
                                                 ¦______________________¦
                                                     8        4     4
                                                 ------------------------
   IM       ORIM         OR    ##DATA,RD         ¦  4A    ¦ RD   ¦ 8    ¦
                                                 ¦________¦______¦______¦
                                                 ¦        DATA          ¦
                                                 ¦______________________¦
```

Description:

The Derived Operand is bit-by-bit inclusively ORed with the contents of RD. The result is stored in register RD. The CS is set based on the result in register RD.

POPM

Pop multiple registers off the stack.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	POPM	POPM RS,RD	8F (8) | RD (4) | RS (4)

Description:

For RD<=RS, registers RD through RS are loaded sequentially from a stack in memory using R15 as the stack pointer. For RD>RS, registers RD through R14 and then R0 through RS are loaded sequentially from the stack. In both cases, as each word is popped from the stack, R15 is incremented by one; if R15 is included in the transfer, it is effectively ignored; and on completion R15 points to the top word of the remaining stack.

PUSHM

Push multiple registers onto the stack.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	PSHM	PUSHM	RS,RD	9F	RD	RS

Description:

For RD<=RS, the contents of RS through RD are pushed onto a stack in memory using R15 as the stack pointer. As each register contents are pushed onto the memory stack, R15 is decremented by one word for each word pushed. On completion, R15 points to the last item on the stack, the contents of RD. For RD>RS, the contents of RS through R0, and then the contents of R15 through RD, are pushed onto the stack. On completion, R15 points to the last item on the stack, the contents of RD. In both cases, successively increasing addresses on the stack correspond to successively increasing register addresses.

RET

Unstack IC and return from subroutine.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	URS	RET	@+RS	7F	RS	0

Description:

The contents of the memory location pointed to by register RS is loaded into the instruction counter. RS is then incremented by one. Any one of the 16 general registers may be designated as the stack pointer. This instruction is the subroutine return for the CALL instruction.

ROL

Shift left cyclic.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
				8 / 4 / 4
R	SLC	ROL	#N,RD	| 63 | N-1 | RD |
				1<=N<=16

Description:

The contents of register RD are shifted left cyclically N positions. The shifted result is stored in RD. The cyclic left shift operation is as follows: bits shifted out of the sign bit position (bit 0) enter the least significant bit position (bit 15) and consequently, no bits are lost. The condition status, CS, is set based on the result in RD.

ROLL

Double shift left cyclic.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	DSLC	ROLL	#N,RD	68	N-1	RD
				1<=N<=16		

Description:

The concatenated contents of RD and RD+1 are shifted left cyclically N positions. The shifted results are stored in RD and RD+1. The double left shift cyclic operation is as follows: bits shifted out of the sign bit position of RD enter the least significant bit position of RD+1, bits shifted out of the sign bit position of RD+1 enter the least significant position of RD, and, consequently, no bits are lost. The condition status, CS, is set based on the result in RD and RD+1.

ROLR

Shift cyclic, count in register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	SCR	ROLR	RS,RD	8 / 4 / 4: 6C | RD | RS ; |(RS)|<=16

```
  8       4     4
 ----------------------
 ¦ 6C   ¦  RD ¦  RS ¦
 ¦______¦_____¦_____¦
     ¦(RS)¦<=16
```

Description:

The contents of register RD are shifted cyclically N positions, where N is the contents of register RS. If N is positive, then the shift direction is left; if N is negative (2's complement notation), then the shift direction is right. The condition status is set based on the result in RD.

ROLRL

Double shift cyclic, count in register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	DSCR	ROLRL	RS,RD	8 / 4 / 4: 6F | RD | RS; |(RS)|<=32

Description:

The concatenated contents of registers RD and RD+1 are shifted cyclically N positions, where register RS contains the count, N. If the count is positive, the shift direction is left. If the count is negative (2's complement notation) the shift direction is right. The condition status, CS, is set based on the result in RD and RD+1.

SETI

Set bit.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	SBR	SETI	#N,RD	8: 51 | 4: N | 4: RD
D	SB	SETI	#N,ADDR	8: 50 | 4: N | 4: RX / ADDR
DX	SB	SETI	#N,ADDR(RX)	
I	SBI	SETI	#N,@ADDR	8: 52 | 4: N | 4: RX / ADDR
IX	SBI	SETI	#N,@ADDR(RX)	
R	SVBR	SETI	RS,RD	8: 5A | 4: RD | 4: RS

Description:

The designated bit of the Derived Operand is set to one.

SHL

Shift left logical.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	SLL	SHL	#N,RD	60	N-1	RD
				1<=N<=16		

Description:

The contents of register RD are shifted left logically N positions. The shifted result is stored in RD. The logical shift left operation is as follows: zeroes enter the least significant bit position (bit 15) and bits shifted out of the sign bit position (bit 0) are lost. The condition status, CS, is set based on the result in register RD.

SHLL

Double shift left logical.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	DSLL	SHLL	#N,RD	

8	4	4
65	N-1	RD

1<=N<=16

Description:

The concatenated contents of RD and RD+1 are shifted left logically N positions. The shifted results are stored in RD and RD+1. The double left shift logical operation is as follows: zeroes enter the least significant bit position of RD+1, bits shifted out of the sign position of RD+1 enter the least significant bit of RD and bits shifted out of the sign position of RD are lost. The condition status, CS, is set based on the result in registers RD and RD+1.

SHLR

Shift logical, count in register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	SLR	SHLR	RS,RD	see below

```
      8          4      4
 ----------------------------
 |   6A    |   RD  |   RS  |
 |_________|_______|_______|
         |(RS)|<=16
```

Description:

The contents of register RD are shifted logically N positions, where N is defined as the contents of register RD. If N is positive then the shift direction is left; if N is negative then the shift direction is right. The condition status, CS, is set based on the result in RD.

SHLRA

Shift arithmetic, count in register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	SAR	SHLRA	RS,RD	8 / 4 / 4: ¦ 6B ¦ RD ¦ RS ¦ ; ¦(RS)¦<=16

Description:

The contents of register RA are shifted arithmetically N positions, where N is defined as the contents of register RS. If N is positive, then the shift direction is left; if N is negative (2's complement notation), then the shift direction is right. The condition status, CS, is set based on the result in RA.

SHLRAL

Double shift arithmetic.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	DSAR	SHLRAL RS,RD	see below

```
     8         4      4
 ---------------------------
 |   6E    |  RD  |  RS   |
 |_________|______|_______|
       |(RS)|<=32
```

Description:

The concatenated contents of registers RD and RD+1 are shifted arithmetically N positions where register RS contains the count. If the count is positive, then the shift direction is left. If the count is negative (2's complement notation), then the shift direction is right. The CS is set based on the result in RD and RD+1.

SHLRL

Double shift logical, count in register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	DSLR	SHLRL	RS,RD	8 / 4 / 4 ¦ 6D ¦ RD ¦ RS ¦ ¦(RS)¦<=32

Description:

The concatenated contents of registers RD and RD+1 are shifted logically N positions where register RS contains the count. If the count is positive, then the shift direction is left. If the count is negative (2's complement notation), then the shift direction is right. The condition status, CS, is set based on the result in RD and RD+1.

SHR

Shift right logical.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
R	SRL	SHR	#N,RD	see below

```
     8         4       4
 ---------------------------
 |  61    |  N-1  |   RD   |
 |________|_______|________|
        1<=N<=16
```

Description:

The contents of register RD are shifted right logically N positions. The shifted result is stored in RD. The logical shift right operation is as follows: zeroes enter the sign bit position (bit 0) and bits shifted out of the least significant bit position (bit 15) are lost. The condition status, CS, is set based on the result in register RD.

SHRA

Shift right arithmetic.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	SRA	SHRA	#N,RD	62	N-1	RD
					1<=N<=16	

Description:

The contents of register RD are shifted right arithmetically N positions. The shifted result is stored in RD. The arithmetic right shift operation is as follows: the sign bit, which is not changed, is copied into the next position for each position shifted and bits shifted out of the least significant bit position (bit 15) are lost. The condition status, CS, is set based on the result in register RD.

SHRAL

Double shift right arithmetic.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	DSRA	SHRAL #N,RD	67 (8) | N-1 (4) | RD (4); 1<=N<=16

Description:

The concatenated contents of RD and RD+1 are shifted right arithmetically N positions. The shifted results are stored in RD and RD+1. The double shift right arithmetic operation is as follows: the sign bit of RD, which is not changed, is copied into the next position for each position shifted, bits shifted out of the least significant position of RD enter the sign bit position of RD+1, and bits shifted out of the least significant bit position of RD+1 are lost. The condition status, CS, is set based on the result in register RD and RD+1.

SHRL

Double shift right logical.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	DSRL	SHRL	#N,RD	66	N-1	RD
					1<=N<=16	

Description:

The concatenated contents of RD and RD+1 are shifted right logically N positions. The shifted results are stored in RD and RD+1. The double logical right shift operation is as follows: zeroes enter the sign bit position of RD, bits shifted out of the least significant bit position of RD enter the sign bit position of RD+1 and bits shifted out of the least significant bit position of RD+1 are lost. The condition status, CS, is set based on the result in register RD and RD+1.

ST

Single precision store.

```
Addr      Mil      Assembler Language        Instruction Format
Mode      Std           Syntax
                                               4    2    2    8
                                             ----------------------
 B        STB       ST    R2,DSPL(BR)        |  0  | 2| BR'|DSPL|
                                             |_____|__|____|____|

                                              4    2    2    4    4
                                             --------------------------
 BX       STBX      ST    R2,(BR,RX)         | 4 | 0 | BR' | 2 | RX |
                                             |___|___|_____|___|____|

                                                8       4      4
                                             ----------------------
 D        ST        ST    RS,ADDR            |  90   |  RS  |  RX  |
                                             |_______|______|______|
 DX       ST        ST    RS,ADDR(RX)        |        ADDR         |
                                             |_____________________|
                                                8       4      4
                                             ----------------------
 I        STI       ST    RS,@ADDR           |  94   |  RS  |  RX  |
                                             |_______|______|______|
 IX       STI       ST    RS,@ADDR(RX)       |        ADDR         |
                                             |_____________________|
                                                8       4      4
                                             ----------------------
 D        STC       ST    #DATA,ADDR         |  91   | DATA |  RX  |
                                             |_______|______|______|
 DX       STC       ST    #DATA,ADDR(RX)     |        ADDR         |
                                             |_____________________|
                                                8       4      4
                                             ----------------------
 I        STCI      ST    #DATA,@ADDR        |  92   | DATA |  RX  |
                                             |_______|______|______|
 IX       STCI      ST    #DATA,@ADDR(RX)    |        ADDR         |
                                             |_____________________|
```

Description: The contents of the source operand are stored into the destination operand.

STB

Store byte.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
D	STLB/STUB	STB RbS,ADDR	8: 9C/9B; 4: RS; 4: RX / ADDR
DX	STLB/STUB	STB RbS,ADDR(RX)	
I	SLBI/SUBI	STB RbS,@ADDR	8: 9E/9D; 4: RS; 4: RX / ADDR
IX	SLBI/SUBI	STB RbS,@ADDR(RX)	

Description:

The lower byte of the source register is stored into the designated byte of the destination. The other byte remains unchanged.

STL

Double precision store.

```
 Addr        Mil      Assembler Language       Instruction Format
 Mode        Std           Syntax

                                                4    2    2    8
                                              -------------------
   B         DSTB       STL    RO,DSPL(BR)    |  0  | 3| BR'|DSPL|
                                              |_____|__|____|____|

                                               4    2    2    4    4
                                              ------------------------
   BX        DSTX       STL    RO,(BR,RX)     | 4 | 0 | BR' | 3 | RX |
                                              |___|___|_____|___|____|

                                                 8      4     4
                                              -------------------
   D         DST        STL    RS,ADDR        |  96  |  RS |  RX |
                                              |______|_____|_____|
   DX        DST        STL    RS,ADDR(RX)    |      ADDR        |
                                              |__________________|

                                                 8      4     4
                                              -------------------
   I         DSTI       STL    RS,@ADDR       |  98  |  RS |  RX |
                                              |______|_____|_____|
   IX        DSTI       STL    RS,@ADDR(RX)   |       ADDR       |
                                              |__________________|
```

Description:

The contents of registers RS and RS+1 are stored into the Derived Address, DA and DA+1, respectively.

STM

Store multiple registers.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
D	STM	STM R0,ADDR,#N	99 (8) | N (4) | RX (4)
DX	STM	STM R0,ADDR(RX),#N	ADDR

Description:

The contents of register R0 are stored into the Derived Address, then the contents of R1 are stored into DA+1. The contents of register RN are stored into DA+N where N is an integer, 0<=N<=15. Effectively this instruction allows the transfer of (N+1) words from the register file to memory.

STMI

Store register through mask.

Addr Mode	Mil Std	Assembler Language Syntax	
D	SRM	STMI	RS,ADDR
DX	SRM	STMI	RS,ADDR(RX)

Instruction Format

```
      8        4      4
 ---------------------------
 |  97    |   RS  |  RX   |
 |________|_______|_______|
 |          ADDR          |
 |________________________|
```

Description:

The contents of register RS are stored into the Derived Address through the mask in register RS+1. For each position in the mask that is a one, the corresponding bit of register RS is stored in the corresponding bit of the Derived Address. For each position in the mask that is a zero, no change is made to the corresponding bit of the Derived Address.

STX

Extended precision floating point store.

Addr Mode	Mil Std	Assembler Language Syntax	
D	EFST	STX	RS,ADDR
DX	EFST	STX	RS,ADDR(RX)

Instruction Format

8	4	4
9A	RS	RX
ADDR		

Description:

The contents of registers RS, RS+1, and RS+2 are stored at the Derived Address, DA, DA+1 and DA+2.

SUB

Single precision integer subtract.

```
Addr    Mil       Assembler Language       Instruction Format
Mode    Std           Syntax
                                              8      4     4
                                           ------------------
 R      SR        SUB   RS,RD              |  B1  | RD  | RS  |
                                           |______|_____|_____|
                                             4   2    2     8
                                           ------------------
 B      SBB       SUB   DSPL(BR),R2        |  1  | 1| BR' |DSPL|
                                           |_____|__|_____|____|

                                             4   2    2     4    4
                                           -----------------------
 BX     SBBX      SUB   (BR,RX),R2         | 4 | 0 | BR' | 5 | RX |
                                           |___|___|_____|___|____|

                                              8      4     4
                                           ------------------
 D      S         SUB   ADDR,RD            |  BO  |  RD |  RX |
                                           |______|_____|_____|
 DX     S         SUB   ADDR(RX),RD        |       ADDR       |
                                           |__________________|
                                              8    4     4
                                           ------------------
 IM     SISP      SUB   #DATA,RD           |  B2 | RD |DATA-1|
                                           |_____|____|______|
                                              1<=DATA<=16

                                              8      4     4
                                           ------------------
                                           |  4A  |  RD |  2  |
 IM     SIM       SUB   ##DATA,RD          |______|_____|_____|
                                           |      DATA        |
                                           |__________________|
                                              8      4     4
                                           ------------------
 D      DECM      SUB    #DATA,ADDR        |  B3  |DATA-1| RX |
                                           |______|______|____|
 DX     DECM      SUB    #DATA,ADDR(RX)    |      ADDR        |
                                           |__________________|
                                              1<=DATA<=16
```

Description: The Source Operand is subtracted from the contents of the destination operand. The result, a 2's complement difference, is stored in the destination operand. The condition status, CS, is set based on the result and carry. A fixed point overflow occurs if both operands are of opposite sign and the sign of the derived operand is the same as the sign of the difference.

SUBF

Floating point subtract.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	FSR	SUBF RS,RD	8 / 4 / 4: B9 | RD | RS
B	FSB	SUBF DSPL(BR),R0	4 / 2 / 2 / 8: 2 | 1 | BR' | DSPL
BX	FSBX	SUBF (BR,RX),R0	4 / 2 / 2 / 4 / 4: 4 | 0 | BR' | 9 | RX
D	FS	SUBF ADDR,RD	8 / 4 / 4: B8 | RD | RX; ADDR
DX	FS	SUBF ADDR(RX),RD	

Description:

The floating point Derived Operand is the floating point subtracted from the contents of registers RD and RD+1. The result is stored in registers RD and RD+1. The process of this operation is as follows: the mantissa of the number with the smaller algebraic exponent is shifted right and the exponent incremented by one for each bit shifted until the exponents are equal. The mantissa of the Derived Operand is then subtracted from (RD,RD+1). If the difference overflows the 24-bit mantissa then it is shifted right one position, the sign bit restored, and the exponent incremented by one. If the exponent exceeds HEX(7F) as a result of this incrementation overflow occurs and the operation is terminated. If the sum does not result in exponent overflow, the result is normalized. If during the normalization process the exponent is decremented below HEX(80) then underflow occurs and a zero is inserted for the result.

SUBL

Double precision integer subtract.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	DSR	SUBL RS,RD	8: B7, 4: RD, 4: RS
D	DS	SUBL ADDR,RD	8: B6, 4: RD, 4: RX / ADDR
DX	DS	SUBL ADDR(RX),RD	

```
   8      4     4
 ---------------------
 | B7  | RD  | RS  |
 |_____|_____|_____|

   8      4     4
 ---------------------
 | B6  | RD  | RX  |
 |_____|_____|_____|
 |      ADDR       |
 |_________________|
```

Description:

The double precision Derived Operand is subtracted from the contents of registers RD and RD+1. The result, a 2's complement 32-bit difference, is stored in registers RD and RD+1. The MSH is RD. The condition status, CS, is set based on the double precision results in RD and RD+1 and carry. A fixed point overflow occurs if both operands are of opposite sign and the sign of the derived operand is the same as the sign of the difference.

SUBX

Extended precision floating point subtract.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	EFSR	SUBX RS,RD	8: BB, 4: RD, 4: RS
D	EFS	SUBX ADDR,RD	8: BA, 4: RD, 4: RX; ADDR
DX	EFS	SUBX ADDR(RX),RD	

```
   8       4     4
 ------------------
 |  BB  | RD  | RS  |
 |______|_____|_____|

   8       4     4
 ------------------
 |  BA  | RD  | RX  |
 |______|_____|_____|
 |      ADDR        |
 |__________________|
```

Description:

The extended precision floating point Derived Operand is extended floating point subtracted from the contents of registers RD, RD+1, and RD+2. The result is stored in registers RD, RD+1, and RD+2. The process of this operation is as follows: the mantissa of the number with the smaller algebraic exponent is shifted right and the exponent is incremented by one for each bit shifted. When the exponents are equal, the mantissas are subtracted. If the difference overflows the 39-bit mantissa, then the difference is shifted right one position, the sign bit restored, and the exponent is incremented. If the exponent exceeds HEX(7F) as a result of this incrementation, overflow occurs and the operation is terminated. If the sum does not result in exponent overflow, the result is normalized. If during the normalization process the exponent is decremented below HEX(80), then overflow occurs and a zero is inserted for the result.

TESTI

Test bit.

```
 Addr       Mil      Assembler Language        Instruction Format
 Mode       Std          Syntax

                                                    8      4      4
                                                 ----------------------
   R        TBR        TESTI #N,RD               |  57   |  N   | RD   |
                                                 |_______|______|______|

                                                    8      4      4
                                                 ----------------------
   D        TB         TESTI #N,ADDR             |  56   |  N   |  RX  |
                                                 |_______|______|______|
   DX       TB         TESTI #N,ADDR(RX)         |        ADDR         |
                                                 |_____________________|

                                                    8      4      4
                                                 ----------------------
   I        TBI        TESTI #N,@ADDR            |  58   |   N  |  RX  |
                                                 |_______|______|______|
   IX       TBI        TESTI #N,@ADDR(RX)        |        ADDR         |
                                                 |_____________________|

                                                    8      4      4
                                                 ----------------------
   R        TVBR       TESTI RS,RD               |  5E   |  RD  | RS   |
                                                 |_______|______|______|
```

Description:

Bit number N (0<=N<=15) of the Derived Operand, DO (or the lower order 4 bits of the source register), is tested. Then the Condition Status, CS, is set to indicate non-zero if bit number N of the DO contains a 1. Otherwise CS is set to indicate zero. The MSB of the DO is designated bit number zero and the LSB of the DO is designated bit number 15.

TESTSETI

Test and set bit.

Addr Mode	Mil Std	Assembler Language Syntax
D	TSB	TESTSETI #N,ADDR
DX	TSB	TESTSETI #N,ADDR(RX)

Instruction Format

```
       8        4      4
 ---------------------------
 |  59    |   N   |   RX   |
 |________|_______|________|
 |          ADDR           |
 |_________________________|
```

Description:

Bit number N (0<=N<=15) of the Derived Operand is tested and set to 1. The CS is set according to the test.

XCH

Exchange words in register.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format		
				8	4	4
R	XWR	XCH	RS,RD	ED	RD	RS

Description:

The contents of register RD are exchanged with the contents of register RS. The CS is set based on the result in register RD.

XCHB

Exchange bytes in register.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
			8 / 4 / 4
S	XBR	XCHB RS	EC | RS | 0

Description:

The upper byte of register RS is exchanged with the lower byte of register RS. The CS is set based on the result in register RS.

XIO

Execute Input/Output.

Addr Mode	Mil Std	Assembler Language Syntax
IM	XIO	XIO RS,#CMD
IMX	XIO	XIO RS,#CMD(RX)

Instruction Format

```
      8      4      4
 ----------------------
 :  48  :  RS  :  RX  :
 :______:______:______:
 :        CMD         :
 :____________________:
```

<u>Description:</u>

The input/output instruction transfers data between an external/internal device and the register RS. The derived operand specifies the operation to be performed or the device to be addressed. The immediate operand field may be viewed as an operation code extension field. Note that if indexing is specified, then the input/output operation or device address is formed by summing the contents of the register RX and the immediate field. This is a privileged instruction.

XIOM

Vectored Input/Output.

Addr Mode	Mil Std	Assembler Language Syntax		Instruction Format
D	VIO	XIOM	RS,ADDR	
DX	VIO	XIOM	RS,ADDR(RX)	

```
      8        4       4
 ---------------------------
 |  49    |  RS    |  RX   |
 |________|________|_______|
 |          ADDR           |
 |_________________________|
```

Description:

The input/output operation or device address is specified by the sum of the CMD and the product of the bit number of the bit set in the vector times the contents of RS. This device address is then interpreted as specified by the XIO instruction with the exception that I/O data is transferred to or from DA+2+i rather than RS (where i starts at zero and is incremented after each transfer). This is a privileged instruction.

XOR

Exclusive logical or.

Addr Mode	Mil Std	Assembler Language Syntax	Instruction Format
R	XORR	XOR RS,RD	8 / 4 / 4: | E5 | RD | RS |
D	XOR	XOR ADDR,RD	8 / 4 / 4: | E4 | RD | RX | ADDR |
DX	XOR	XOR ADDR(RX),RD	
IM	XORM	XOR ##DATA,RD	8 / 4 / 4: | 4A | RD | 9 | DATA |

Description:

The Derived Operand is bit-by-bit exclusively ORed with the contents of RD. The result is stored in RD. The condition status, CS, is set based on the result in RD.

```
@copy m1750a.mem tty:,
@@byte 8
@@
 M1750A.MEM.78 => TTY:M1750A
```

dummy

DUAL USER'S MANUAL

DYNAMIC UNIVERSAL ASSEMBLY LANGUAGE

VERSION 82

1 APRIL 1982

PROPRIETARY SOFTWARE SYSTEMS, INC.
9911 West Pico Boulevard, Penthouse K
Los Angeles, California 90035
213/553-2997

PREVIOUS PAGE IS BLANK

TABLE OF CONTENTS

Table of Contents (continued)

Table of Contents - continued

Table of Contents - continued

Table of Contents - continued

CHAPTER 1

INTRODUCTION

The need for effective communication between man and computer in current man/machine systems can only be satisfied via the use of highly discriminating programming languages. These languages must be capabable of expressing the most intricate problem in a lucid, brief and meaningful form.

Ideally, a programming language should be easy to learn, machine independent, and adaptable to the problems of business, government, science and engineering. Currently, nearly all programming languages fall into two basic categories:

1. Machine oriented languages

2. Problem or procedural oriented languages

Neither of these two languages contain the power or flexibility required for the solution of the wide variety of problems being solved by modern computer systems.

The purpose of DUAL is to bridge the gap between machine and procedure oriented languages. Using DUAL, a programmer has the necessary tool for effective solution to current and future computing problems.

DUAL User's Manual describes the Dynamic Universal Assembly Language and the characteristics of the language which is translated by the DUAL processor into relocatable programs. DUAL represents a considerable extension with respect to traditional macro or meta assemblers. A meta assembler is a twofold tool by its very nature. It is a powerful assembly or machine language processor, and it also contains directives which enable the user to define and implement a higher order (procedural or English-like) language tailored to a specific application. The DUAL processor translates a set of directives into code or data suitable for execution on a target computer. The directives found in DUAL include those present in most comprehensive assemblers plus a special set of META-Directives used to define computer instructions, data formats and environmental conditions.

DUAL accepts words, statements and phrases to produce machine language instructions. It is more than an assembly program because it has compiler-like facilities. Via DUAL METAs, a coding system can be developed that parametrically translates a single phrase into groups of computer instructions (for example, a FORTRAN-like compiler could easily be implemented using DUAL). DUAL provides both the system and application programmer a powerful set of directives whereby he can reduce programming time, improve program checkout and develop languages to meet the needs of a specific application.

DUAL is currently operational on a wide variety of host computers, including:

1. IBM 360/370 (OS,DOS,CMS,VS)
2. RCA Spectra 70 (DOS)
3. DEC-10 (TOPS)
4. UNIVAC 1100 series (Exec 2 and Exec 8)
5. DEC PDP-11 (RT11 and DOS)
6. GE-635 (GECOS)
7. Interdata 7/32 and 8/32 (MT3)
8. CDC 6000 series (KRONOS and SCOPE)
9. Varian V75 (DOS)
10. Xerox Sigma 5/7/9 (BPM,CP-V)
11. Honeywell Level 66DPS (CP-6)
12. VAX-780 (VMS)

Included as part of the DUAL system is a set of built-in cross assemblers, including

1. Digital Equipment PDP-11
2. Data General NOVA
3. Intel 8080
4. Hewlett Packard 2100
5. IBM 360, AP-1
6. Univac ANUYK 7,1230
7. Xerox Sigma 5, 930
8. Honeywell 116-716
9. Teledyne TDY-43
10. MIL-STD 1750A
11. Honeywell Level 66DPS and Level 6
12. Zilog Z8001,Z8002

CHAPTER 2

GLOSSARY

Elements and terminology of the DUAL system are provided here that the reader may skim this section with the idea of using it later as a reference. The information within the glossary is presented alphabetically.

2.1 A solute

A program that can be loaded into memory at only one precise starting location.

2.2 Address

A designation indicating the location of information in a computer memory.

2.3 Argument Field

The argument field consists of a series of subfields, each being an argument, separated by commas. The argument field begins with the first non-blank after the command field, and terminates with the first blank that is not within parentheses.

2.4 ASCII

American Standard Code for Information Interchange.

2.5 Assembler

A program which translates symbolic commands and tags to an appropriate numeric pattern. Usually an assembler will have a fixed number of commands that it can translate for a particular computer.

2.6 Balance

An expression is considered "BALANCED" if it never has more than one relocatable element unless the relocatable elements are separated by a minus sign.

2.7 Base

Total number of distinct marks used in a numbering system. A quantity used implicitly to define a particular system of representing numbers by positional notation. Also used to describe address registers in a page computer.

2.8 Binary

The binary number system uses 2 as its base of notation. Numbers consist of the digits 0 and 1. In the binary number system, numbers are represented as the sum of powers of 2.

2.9 Binary Code

A numeric representation of object which can directly be loaded into memory.

2.10 Bit

A binary digit.

2.11 Blank

A character represented by a space.

2.12 Command Field

The field that begins after the label field and usually contains a DUAL directive, META reference or a machine instruction.

2.13 Comments

DUAL allows the user to write explanatory remarks on input statements. These explanatory messages are not processed by DUAL and can either appear after the last general field or in an entire input statement that has an "*" in column 1.

2.14 Compiler

A program which translates symbolic higher level (English, mathematical notation, etc.) language elements into machine instructions.

2.15 Configuration

An assembly of machines, devices, systems, etc., that work together.

2.16 Constant

A constant is an unchanging quantity (not a variable).

2.17 Cumulative Meta Nesting Level

The sum of the nesting levels of all previous meta calls. This is used to generate unique symbols.

2.18 Current Location Counter

The present value of the symbol "$". This is the next location where DUAL will generate object code.

2.19 Data Processing

The handling, storage and analysis of information in a sequence of systematic and logical operations by a computer.

2.20 Direct Reference Address

The location being accessed.

2.21 Directive

An order (or command) to the DUAL Processor to perform a specific operation.

2.22 DUAL

Acronym for Dynamic Universal Assembly Language.

2.23 DUAL Processor

A computer program which translates Dynamic Universal Assembly Language source input statements to relocatable or absolute object code.

2.24 Evaluatable Expression

A legal expression which contains operands that have been previously defined or are constants.

2.25 Execution Time

That which occurs when the assembled program is operating.

2.26 Expression

A series of terms separated by operators that combine to form a value.

2.27 External Definition

An external definition defines a symbol in one program that can be referenced in another.

2.28 External Reference

An external reference allows the use of symbols that were defined in another program to be utilized in this program.

2.29 Field

A consecutive group of non-blank columns terminated by a blank that is not within parentheses. An input statement in DUAL is broken down into fields. The first three fields are the label, command and argument field.

2.30 Field Function

A function that references a subfield (e.g., NC,LF,CF,GF,AF, C#,VS,SC,SV).

2.31 FORTRAN

A procedural programming language with extensive arithmetic facilities. FORTRAN is an abbreviation of FORmula TRANslation originally developed by International Business Machines.

2.32 Forward Reference

A symbol used in an expression which has not previously been defined is a forward reference.

2.33 Function

DUAL has a set of intrinsic (built-in) operations which are termed functions. Certain functions are used in generating specific types of constants, while others are used for scanning arguments from a META call line within a META definition.

2.34 Functional Reference

The following group of functions are considered to be the "functional references": AF,CF,LF,AFA,CFA,OP

2.35 General Field

Any field in DUAL may be referred to as a specific general field. The first three fields on a statement have alternate names (Label, Command, Argument) but the remaining fields can be referred to only with general field terminology. The default option is three general fields. See Length directive for further information.

2.36 Global Symbol

A global symbol is a symbol that can be referenced anywhere in a program (as opposed to a local symbol). The first character of a global symbol must be alphabetic.

2.37 Hardware

The physical devices used in a computer system.

2.38 Hexadecimal

The hexadecimal number system uses 16 as its base of notation. Numbers consist of the following characters: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

2.39 Indirect Addressing

The initial address contains the direct reference address.

2.40 Input Statement

An input statement is comprised of one or more 80 character records from the source device.

2.41 Instruction

A machine word or a language symbol that directs a computer to take a certain action.

2.42 Intrinsic Function

Any of the functions that exist in DUAL are called intrinsic functions. That is, they are built in to the DUAL processor.

2.43 I/O

An abbreviation for Input/Output.

2.44 Iterative Loop

A means by which a desired computer operation is performed repeatedly.

2.45 Label

A symbol appearing in the label field.

2.46 Label Field

The initial field on an input statement. The label field must begin in column 1 of an input statement.

2.47 Literal

An expression enclosed in parentheses with a leading "L". Literals provide a convenient method for introducing data into a program. The value of the expression is placed in the literal pool, and the expression results in a forward reference to be satisfied at the end of the program.

2.48 Literal Pool

A table which contains the values of the literals which have been processed by DUAL during the translation of a program.

2.49 Loader

A program which translates object code into a format which can be directly executed in a specific computer.

2.50 Local Symbol

A symbol that begins with a $ and is two or more characters is defined as a local symbol. The symbol "$" used in a source line references the current value of the location counter and is not a local symbol.

2.51 Logical Device

A logical device denotes an input/output function as opposed to a physical device (which denotes an actual piece of hardware, e.g., card punch). The following are logical devices in DUAL:

- Source input
- Listing output
- Symbolic output
- Object output
- Update input
- Update output
- Library input
- Binary input
- Operator communication
- Intermediate output
- Intermediate input
- System object

2.52 Machine List

A series of positive values separated by commas, whose total is less than or equal to the target computer's word size. A machine list is equated to a symbol via the LIST directive.

2.53 Machine Instruction

A specific command to be performed by a target computer.

2.54 Macro Instruction

A source statement that is usually translated to several instructions in machine language.

2.55 META/META Directives

DUAL provides directives which enable the user to conditionally determine the order of input statement processing. These directives are termed META directives (e.g., JUMPSYM, JUMPVAL, LOOP, WITHHOLD, etc.). For further explanation see Section 7.

2.56 META Call Line

An input statement whose initial command field contains the name of a META call line will cause the processing of the statements within the associated META definition.

2.57 META Definition

A series of input statements bounded by a META and MEND statement form a META definition. A META definition can have input statements that use (scan) the arguments from the META call line to conditionally generate the desired object code or values.

2.58 META Language

A language used to explain or describe another language.

2.59 Numerical String

A string of decimal characters.

2.60 Object Code

DUAL converts the source input statements to a format that is easily loaded into a computer memory. This format is called object. Object code differs from binary code in that it contains the necessary information to allow for relocation and program inter-communication.

2.61 Octal

The octal number system uses 8 as its base of notation. Numbers consist of the following digits: 0,1,2,3,4,5,6,7.

2.62 Operation Code (OP)

The part of an instruction designating the operation to be performed.

2.63 Operator

An operator describes a particular arithmetic, relational, or logical operation. DUAL allows the following operators:

+	addition
-	subtraction
/	division
*	multiplication
**	binary shift
.OR.	logical or
.EOR.	logical exclusive or
.AND.	logical and
.NOT.	logical complement
.EQ.	equality
.NQ.	inequality
.GT.	greater than
.GQ.	greater or equal
.LT.	less than
.LQ.	less or equal

2.64 Parameter

A symbol which has been equated to a value is termed a parameter.

2.65 Processor

The computer program which translates source input statements to object code.

2.66 Program

A series of source input statements terminated by an END statement. A program is a sequence of instructions which form a step-by-step process for solving a problem with a computer.

2.67 Programmer

A person who designs, codes, debugs and documents a computer program.

2.68 Pseudo Operation

This term is synonymous with directive.

2.69 Recursive META

A META which calls itself either directly or indirectly by calling other METAs which call it in turn is said to be recursive.

2.70 Relational

A symbol that is used to compare two values or character strings. In DUAL the following symbols are relationals (EQ,GQ,GT,LQ,LT,NQ).

2.71 Relocatable

A program that can be loaded into memory at more than one starting location. Relocatable locations within a program are relative to the initial location.

2.72 Reserved Symbol

A symbol that has a fixed interpretation by DUAL and is considered part of the DUAL system.

2.73 Skip to Statement

A statement which causes DUAL to ignore subsequent input statements until a specified symbol is found in the label field of a subsequent input statement.

2.74 Software

The programs which direct the operations of a computer.

2.75 Source Input Statement

The basic unit of input to the DUAL processor.

2.76 Source Program

Symbolic representation of a program.

2.77 Special Characters

DUAL considers the following as special characters: + - / ' () . , "blank" * $ # @ _

2.78 Subfield

A portion of a field that begins with a non-blank and is terminated by a comma or the end of the field. Subfields are separated from other subfields by commas that are not within parentheses.

2.79 Subroutine

A series of machine instructions that can be utilized at various points in a program without actually duplicating the instructions.

2.80 Symbol

A string of alpha-numeric characters that can be used to name various elements within a program. A symbol must begin with an alphabetic $,#,@, or _.

2.81 Symbolic Concatenation

A DUAL feature that allows dynamic construction of symbols at processing time.

2.82 Syntax

The rules governing the structure of a language.

2.83 Target Machine

The computer which will execute the object program.

2.84 Term

A term is an element of an expression. It is either a symbol, or a constant or a DUAL function.

2.85 Value

The result obtained from evaluating an expression.

2.86 Void Field

A field that has no non-blank characters is called a void field.

CHAPTER 3

CODING FEATURES OF DUAL

3.1 CHARACTER SET

The basic character set for the DUAL Processor consists of the following characters:

Alphabetic:	A through Z
Numeric:	0 through 9
Special Functional:	+ - * / ( ) . , ' @ # $
	"blank"

The special characters, when appearing in certain contexts, have functional interpretations that are a fixed part of DUAL. In addition to the characters listed above, the DUAL processor will accept any other characters that are available on the computer equipment being employed. For certain usages, only character subsets, as specified in the corresponding discussion, are acceptable.

3.2 SYMBOLS

3.2.1 General

DUAL's symbol capability provides the programmer with a powerful device that reduces programming effort and increases programmer efficiency. With this capability, the user can assign meaningful names to the various elements of his program, and reference those elements by their assigned names. This can result in more readable programs, shorter programs, greater program parameterization and greater process flexibility. A symbol may consist of any number of characters.

3.2.2 Definition

In DUAL, a symbol is a properly constructed string of characters used to represent:

- The location of an instruction
- The location of data
- A parameter or character string
- A list (see LIST directive) or structure (see STRUCTURE directive)
- A source statement at assembly time only
- A META definition
- Terms reserved by the DUAL processor (see Reserved Symbols 3.2.4)

3.2.3 Symbol Construction Rules

Symbols must abide by all of the following rules. If a rule is violated, the DUAL processor will generate an error indication on the program's listing.

1. A symbol must begin with an alphabetic (A-Z), #, @, _, or a $.

2. A symbol can only be composed of alphabetics (A-Z), numerics (0-9), _, @, #, or dollar signs ($). Blanks are not allowed within a symbol.

3. There is no limitation on the number of characters in a symbol (a symbol may be continued from one input statement to another).

The following symbols are valid:

```
$123
$END$OF@PROGRAMS
TIME$TILL$BLAST$OFF
KICKOFFAFTERTOUCHDOWNBYRAMS
LATEST
```

The following are invalid symbols:

```
99ZZ            (begins with numeric)
$A+12B          (+ sign not allowed in symbols)
AB CD           (blank within symbol)
```

3.2.3.1 Local Symbols

A local symbol is defined and recognized only within a specific area of a program (as opposed to global symbols which are recognized throughout the program). Local symbols are used for two purposes (see LOCAL directive):

- To avoid the poblem of duplicate symbols.
- To allow larger programs to be assembled in small memory space.

3.2.3.2 Differentiation of Local and Global Symbols

In DUAL, differentiation between local and global symbols is a simple matter. All local symbols begin with a dollar sign ($).

3.2.4 Reserved Symbols

Certain symbols have fixed interpretations by DUAL and are considered to be part of the DUAL system. These symbols are called reserved symbols. There are five categories of reserved symbols as indicated below:

- DUAL directives
- DUAL relationals
- DUAL functions
- DUAL constants
- $

Command type reserved symbols may be utilized as labels and non-command type reserved symbols may be utilized as command field symbols.

3.3 INPUT STATEMENT DESCRIPTION

3.3.1 General

Input to the DUAL processor consists of a stream of input statements that, in total, comprise the source program. An 80 character logical record as the basic unit of input. With the continuation capability, a single input statement may extend over more than one 80 character record if necessary.

3.3.2 Definition

An input statement is the fundamental element of a program. It begins with the initial coding column of the statement and ends with a statement termination.

In the general case, an input statement consists of three or more fields of information. The first three fields have the following specific names:

- Label field
- Command field
- Argument field

An unlimited number of fields are provided in DUAL. Any field can be referenced as a specifically indexed general field. The rules governing these fields for processing by DUAL are:

1. A field begins in the initial coding column (hereafter referenced as column 1) of the coding sheet or is preceded by a blank.

2. It does not contain any blanks unless they are within parentheses or are part of a field separator.

3. It may continue from one line to the next by its extending to the continuation column and the continuation column being non-blank.

3.3.2.1 Subfield Separator

A subfield separator is used to break a field down into subfields. DUAL has two types of subfield separations; a comma and a quote string (designated by a single quote). A quote string allows comments to be used to separate subfields. For example:

```
ADD A,B,C
```

is exactly the same as:

```
ADD A'TO'B'AND STORE IN'C
```

A quote string at the start of a field is merely ignored. Hereafter, a comma will be used to designate a subfield separator.

3.3.2.2 Subfield

A subfield is a portion of a field that either begins with the initial character of the field or after a comma and ends either with the last character of the field or with a comma (the comma not being within parentheses). The function of commas is to separate subfields. It is important to note that a subfield may contain any combination of characters except blanks or subfield separators that are not within parentheses.

3.3.2.3 Label Field

The label field is so named because it is generally used for assigning a label to the input statement on which it appears. Depending upon the contents of the command field, this may result in the label also being assigned:

1. The value represented by a particular subfield that is part of the input statement.

2. The location of data or the location of an instruction.

3. The symbolic reference of a particular directive specification.

Unless directed otherwise by a COLUMN directive, the label field begins in column 1 and is terminated by the first subsequent blank column that is not within a quote string or parentheses. If a non-blank continuation column is encountered before the end of the label field, the field is continued in the subsequent continuation column of the next line. For most input statements the presence of a label in the label field is optional. A blank or an * in column 1 indicates that no label has been assigned to the input statement. In certain instances, the label field can consist of a series of subfields separated by commas. The same label should not appear in the label field of two different input statements unless the particular label is either:

- A parameter, list, or string redefinition.
- A local symbol.
- A directive where the label field is not processed.

3.3.2.4 Command Field

A command is defined as a DUAL directive, or a mnemonic code that represents a machine instruction, or a META reference. If an input statement does not contain an * in column 1, (see comments), then the command field begins with the first non-blank column after:

- The label, when a label is present
- Column 1, when a label is not present

All input statements other than comment statements must include a command. The command field is ended by a blank that is not within a quote string or parentheses. If a non-blank continuation (see continuation) column is encountered before the end of the command field, the field is continued in the subsequent continuation column of the next line. The command field can consist of a series of subfields separated by commas.

3.3.2.5 Argument Field

An argument is defined as any string of characters appearing in the argument field that does not include any blanks or commas unless they are within parentheses. Most directives, instructions and META utilize one argument field for the necessary arguments.

The presence of the argument field is optional, depending upon the contents of the command field. If the argument field is present, it is terminated by the subsequent appearance of a blank that is not within parentheses. If a non-blank continuation column is encountered before the end of the argument field, the field is continued in the subsequent continuation column of the next line. Generally, the argument field will consist of a series of subfields, each being an argument, separated by commas.

3.3.2.6 General Field

An unlimited number of fields are permitted on a DUAL source statement. Any field can be referenced as a specifically indexed general field. The label, command and argument fields can be synonymously referred to as the first three general fields of a statement.

3.3.2.7 Statement Terminators

Termination of a statement can be indicated in the following ways:

1. A string of blanks of the required length (see LENGTH directive), not within parentheses.

2. A void continuation column.

3. A period not within parentheses, enclosed by blanks.

4. A blank after the last meaningful field.

3.3.3 Comments

Any information on an input statement that is not part of the general fields is treated as comments. In addition, if column 1 of a statement contains an *, or the label and command fields are void, the entire statement is treated as a comment. Generally, comments will consist of explanatory remarks, program identification and sequence numbers.

The inclusion of comments in a source program is entirely at the user's discretion. The comments, if provided, have no effect on the resulting object program. The DUAL processor simply includes the commentary on the output program listing, unless the user has indicated a desire to suppress the listing of input statements.

It should be noted that blanks among comments have not special significance. Likewise, the continuation column does not apply, if comments precede it.

3.3.4 Free-Form Input Statement

With many systems the user is allowed no choice in determining the format of the input statements. DUAL does not require that each field in a statement begin in a specific column and DUAL permits the user to determine the criteria (see LENGTH directive) for terminating a statement. The location of the continuation column (see COLUMN directive) is also under programmer control. The effect of this is that the programmer has greater flexibility and freedom in determining the format of his input statements. For this reason, input statements in DUAL are considered free-form.

Still, certain rules must be adhered to in writing free-form input statements. They are:

1. The label, command and argument fields must appear in the input statements, from left to right, respectively.

2. A label, when present, begins in column 1 unless specified otherwise by a COLUMN directive.

3. All fields are terminated by a blank column (not enclosed in a quote string or parentheses).

4. A string of consecutive blanks equal to the number specified in a LENTH command, or 15 consecutive blanks in the absence of a LENGTH specification, indicates that the remaining information on the input statement is to be considered as comments.

3.4 TERMINATOR

An input statement may be terminated or skipped by using a period. The period must be immediately preceded and followed by at least one blank. In effect, the period terminator causes DUAL to consider the remainder of the input statement as comments.

3.5 CONTINUATION

The user may continue a field from one coding line to the next, and so on without limit, via DUAL's continuation feature. The DUAL continuation mechanism provides two distinct methods for the user to continue a source line over more than one record. The first method is called the "column method." A column is designated (see COLUMN directive) by the user as the continuation column, or lacking a designation from the user, the processor assigns the function to column 73. Thereafter, when the continuation column of an input statement is non-blank, the input statement continues with column 1 of the next line, provided that either:

1. The column that precedes the continuation column is part of a field, or

2. Blanks precede the continuation column. The end of the last field that did appear was within the required length (see LENGTH directive), and a period (.) statement terminator is not present.

However, if the continuation column is blank, the input statement ends with the line being processed. Note that continuation does not apply to comments. The user may redesignate the continuation column as often as necessary.

Any non-blank character may be used to indicate continuation. The character itself is not treated as part of the field that is continued. A source input statement may continue over as many lines as the user deems necessary. The second method utilizes the ; as a continuation indicator. DUAL will automatically skip the rest of the existing source line upon encountering a semicolon(;) and continue processing with the first non-blank of the following statement.

3.6 LOCATION COUNTER

The user may reference the current value of the location counter at any place in his program by using the symbol $. The location counter is used to assign storage addresses to symbols that appear in the label field of certain directives and machine instructions. Symbols may be equated to the location counter via the "EQU" directive (i.e. ALPHA EQU $). Certain directives are provided which enable the user to maintain control over the value of the location counter (see location counter directives).

3.7 CONSTANTS

3.7.1 General

Constants may be presented to the DUAL processor in a variety of representations, hence providing maximum input convenience and flexibility to the user. The value of a constant is not affected by the location in storage of the program and does not change during processing of the source program. The types of constants that can be specified in DUAL and the corresponding rules for usage are described below.

3.7.2 Decimal Integer

A decimal integer is coded as a string of decimal digits with a non-zero leading digit. The maximum number of digits in a decimal integer is determined by the word length of the particular computer being utilized. When stored, the value is right justified.

For example:

1. ALPHA DATA 39 would generate 0000000000100111 at location ALPHA in a 16 bit computer.

2. BETA+15 refers to a location fifteen cells after the location of BETA.

3. DELTA EQU -9 would generate for DELTA the value 111111111111111111110111 in a 24 bit two's complement computer.

3.7.3 Based Integer

A based integer is coded as a string of digits preceded by a zero. The maximum number of digits in a based integer is determined by the word length of the particular computer being utilized. When stored, the value is right justified. For example, CORETEST DATA 0125252 would generate 1010101010101010 in a 16 bit computer (assuming a NUMBASE of 8) and HERE-011 refers to a location 9 decimal cells before the location HERE. The default base value is octal (8). The NUMBASE directive permits the user to specify a base between 2 and 32.

3.7.4 Hexadecimal Integer

A hexadecimal integer is coded as a string of hexadecimal digits (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) enclosed by parentheses and preceded by the term HEX. The maximum number of digits in a hexadecimal integer is determined by the word length of the particular computer being utilized. When stored, the value is right justified. For example:

```
QRSI      EQU      HEX(12AB)     would assign the value
                                 0001 0010 1010 1011
                                 to QRSI in a 16 bit
                                 target computer.
```

3.7.5 Floating Point Number

A floating point constant is constructed as follows:

```
FL( [s] d [E [s] d] [,p] )
```

1. Strings enclosed in brackets [ ] are optional.

2. The s represents a sign indicator: + or -. The default for the sign is positive.

3. The d represents a string of decimal digits (including decimal point).

4. The Esd string represents an exponent. The default for the exponend is E + 0.

5. The p represents the precision of the constant to be created: (1=single precision, 2=double, 3=triple, 4=quadruple). Each precision number used must correspond to a FLOAT directive with the same precision. The default for the precision is one. The maximum precision is four.

The exponent indicates the power of ten by which the number should be multiplied. For example:

- FL(186E3) = +186 x 10(3) = +186000
- FL(18.6E+4) = +18.6 x 10(4) = 18600
- FL(+18.6) = +18.6
- FL(-18.6E-5,2) = -18.6 x 10(-5) = -.000186
  (double precision)
- FL(.186E-1,3) = .186 x 10(-1) = .0186 (triple precision)

3.7.6 Fixed Point Number

A fixed point constant is constructed as follows:

FX ([s] d [E [s] d] [x[s]b] [,p]

1. Strings enclosed in brackets [...] are optional.

2. The s represents a sign indicator: + or -. The default for the sign is +.

3. The d represents a string of decimal digits (including the decimal point).

4. The Esd string represents an exponent. The exponent indicates the power of ten by which the number should be multiplied.

5. The X(F,B,R,T) choice represents a roundoff indicator (R B) or truncation (F T) and two alternative representation methods by number fractions (F R), or by binary position (B T). The b represent a one or two digit decimal number indicating the number of fractional bits. The default for the fraction bit group is F0.

6. The p represents the precision of the constant to be constructed (1=single precision, 2=double, 3=triple, 4=quadruple). Each precision number used must correspond to a FIXED directive with the same precision.

For example:

- FX (.75B10,2)=0....11000(double precision 16 bit word size.)

3.7.7 Binary Constant

A binary constant is coded as a string of binary digits (0 or 1) enclosed by parentheses and preceded by the term BIN. For example:

- BIN(0101010101)
- BIN(111000111000101000)

3.7.8 Double Fixed Point (DBL)

A double fixed point constant is used to generate an integer fraction constant. The function itself returns a value of the fraction as a 32 bit number. The integer result is stored in the DUAL reserved symbol XNC as a 32 bit number. It is the user's responsibility to adjust the value for the desired target computers format. For example:

```
DBL(.7345)
```

3.7.9 Textual Constants

A textual constant is coded as any string of characters enclosed by parentheses and preceded by the term HOL. For example:

- HOL(NOW IS THE TIME)
- HOL(A*ABCD,D./*XYZ*)

The only restriction of the textual string is that parentheses cannot appear within the text. To represent parentheses within a text string, the user utilizes the control character ' (prime). The prime designates one of four possible options (all other cases are illegal):

```
'S is a semicolon
'L is a left parenthesis
'R is a right parenthesis
'' is a single '
```

Note that blanks and commas may appear in a textual string of characters. When stored, as many words of storage are used as indicated by the word length of the target computer. The first character is left justified in the first storage word and trailing blanks are placed in the last storage word if needed.

3.7.10 Textual Constant With Count

The HOLC constant is identical to the above HOL constant except that the initial character contains a character count of the actual number of characters. The size of HOLC character string is limited by the number of bits per character. For example, an 8 bit character target could have a maximum 255 HOLC character string.

The rules for parentheses are the same for HOLC as they are for HOL. For example:

- HOLC(TIME TO WAKE UP)
- HOLC(0123456789ABCDEF ARE HEX NUMBERS)

3.7.11 Character Constants

A character constant is coded as a character enclosed by parentheses and preceded by the letter C. When stored, the character is right justified with leading zeroes. For example:

- C(A)
- C(.)

It should be noted that the rules for parentheses are the same as for textual constants.

3.7.12 Octal Constant

An octal constant is coded as a string of octal digits enclosed in parentheses and preceded by the term OCT. For example:

- OCT(01234567)
- OCT(37777)
- OCT(10)

3.7.13 BCD Constant

A BCD constant is coded as an optional sign followed by a string of decimal digits (0-9) enclosed by parentheses and preceded by the term DEC. The maximum number of digits is determined by the number of bits in the word size of the target computer (4 words worth of data). Each DEC digit is coded as a four bit binary pattern where 0000=0, 1001=9, 1100=+, 1101=-. For Example:

- DEC(12345)
- DEC(-175423)
- DEC(+0123456789876543210 9)

3.7.14 Fraction Constant

A fraction constant generates a fixed point value with the number of bits in the target word size - 1 worth of fractional bits. The format for a fraction constant is:

1. FRAC - required characters indicating fraction constant
2. (- required left parenthesis)
3. s - optional + or - sign
4. i - optional integer string
5. . - optional decimal point
6. f - optional fractional string (either integer and/or fraction must be present.
7. / - required divide symbol
8. s - optional + or - sign
9. i - optional integer string
10. . - optional decimal point
11. f - optional fraction string
12.) - required right parenthesis

The second value must be greater than the first value. The following examples help illustrate the use of the FRAC function:

```
FRAC (73.62/86.4)
FRAC (-19.7435/+86.42)
FRAC (2075.1/-3000)
```

3.8 LITERALS

3.8.1 General

Literals provide a convenient method for introducing data into a program. A literal appearing on an input statement results in its value being stored in the literal pool (see Glossary) and the location's address replacing the corresponding subfield of the input statement.

3.8.2 Definition

In DUAL, a literal is a subfield consisting of an expression enclosed by parentheses and preceded by an L. For example:

- L(123) interpreted as decimal value 123
- L(HEX(ABC)) interpreted as hexadecimal value ABC
- L(BETA+17) interpreted as address BETA+17

Note that by definition, when a literal appears in a subfield, it appears as the only item in the subfield. The result of the evaluation of a literal will be that one of the following will appear in the literal pool:

- The value of the constant.
- The address corresponding to a label.
- The value of a parameter.

3.9 EXPRESSIONS

3.9.1 General

The basic forms of an expression are a single term or a combination of terms separated by operators and possibly including parentheses. Expressions can appear in any subfield where a value is required. The value of a term may be inherent in the term (as with constants) or may be assigned by DUAL (as in the case of a symbol). When DUAL processes an input statement containing an expression, the expression is reduced to a single value, "the value of the expression" and then the input statement is processed using the expression value in place of the expression.

3.9.2 Operators

The operators which can appear between terms of an expression are:

+	for addition
-	for subtraction
*	for multiplication
/	for division
**	for binary shift
.OR.	for logical or
.EOR.	for logical exclusive or
.AND.	for logical and
.NOT.	for logical complement
.EQ.	for relational equal to
.GT.	for relational greater than
.LT.	for relational less than
.NQ.	for relational not equal to
.GQ.	for relational greater than or equal to
.LQ.	for relational less than or equal to

In general, two operators in an expression cannot appear adjacent to one another (** is considered as one operator). The exception is when one of the operators is .NOT. Binary shift is equivalent to multiplying by two to the specified power. For example:

A**B means A times 2(B)

3.9.3 Terms

A term in an expression can be a symbol, a constant or a function. A maximum of two terms that represent addresses may appear in an expression. If address terms appear in an expression, the result must not be a negative address value. A literal cannot appear in a multi-term expression, although it may appear as the term in a single term expression.

3.9.4 Parentheses

Parentheses may be employed in an expression to indicate the desired order of evaluation. Terms within parentheses will be evaluated before the remaining terms of the expression. Parentheses may be nested to any level. In the case of nested parentheses, the innermost terms are evaluated first. Parentheses may also be employed by the user to improve the readability of an expression.

3.9.5 Evaluation Order

An expression consisting of a single term is directly evaluated by assigning the value of the term to the expression. For example:

- 19 results in 19
- SC(A,B) results in the value of symbol AB
- HDLEF results in the value of symbol HDLEF
- FL(1E10) results in the value 10,000,000,000
- HEX(ABC) results in the value 2748

A multi-term expression is evaluated according to the rules stated below:

1. An appropriate value is supplied for each term.

2. Completed sets of parentheses are evaluated from left to right. For example, in the expression A+(B*X)+(D*E), B*X is evaluated before D*E.

3. Arithmetic operations are performed from left to right within an expression level, with the highest level being performed first. The different levels are:

.NOT.	Level 4 (highest)
.EQ.,.NQ.,.GQ.,.LQ.,.LT	Level 3
.GT.,.AND.,.OR.,.EOR.,**	
*,/	Level 2
+,-	Level 1 (lowest)

For example, 3+4*8/2**3 results in the value 5 and is arrived at by first calculating

```
2**3   =  2x2(3) = 16,   then
4*8    =  32,            then
32/16  =  2,             then
3+2    =  5.
```

4. Any remainder resulting from division is dropped, thus division in an expression always yields an integer.

3.9.6 Address Terms

An address term is a symbol to which an address has been assigned. This includes labels on instruction, or DATA or GEN statements or EQU's to another address. Address terms can appear in expressions only in certain formats. Let A represent an address term, and let P represent a parameter term, then the following are legal expressions with addresses:

A	(result Address)
A1-A2	(result Parameter)
A+P	(Address)
A-P	(Address)
A1-A2+P	(Parameter)

Note that if the expression contains address terms, the result must not be a negative address value. In addition, address terms may be combined with other terms only through addition or subtraction and only when the addresses have the same section number. Other attempted combinations with address terms will result in an error notification.

CHAPTER 4

RELATIONALS

Relationals are used for indicating the relational test that should be made between the subfields preceding and subsequent to the appearance of the relational symbol. For example:

- A,GT,5 (is A greater than 5)
- S1,NQ,S2 (is S1 not equal to S2)

The relationals should only appear as a part of the argument field for the JUMPSYM and JUMPVAL directives. When a relational symbol is used, it must appear alone in its designated subfield. Relationals can be used within META definitions to compare an argument from the META call statement against a value or character string, and thereby determine the sequence of input statement processing. The DUAL relational symbols and their interpretations appear below:

- EQ equal to, (=)
- GQ greater than or equal to, (>=)
- GT greater than, (>)
- LQ less than or equal to, (<=)
- LT less than, (<)
- NQ not equal to, (<>)

CHAPTER 5

FUNCTIONS

5.1 INTRODUCTION

DUAL's functions provide a mechanism for scanning character strings, designating arguments, manipulating character strings, and retrieving values. Certain functions are part of the mechanism for relating subfields of a "META call" to references within the META definitions. Other functions indicate the number of subfields, or the DUAL classification of a symbol, or the existence of an indirect (*) reference, or are used to generate specific types of constants.

5.2 FIELD DESIGNATORS (LF,CF,AF,GF,EF)

The label, command, argument and general field functions (LF,CF,AF,GF respectively) have the following general formats:

- LF(subfield number)
- CF(subfield number)
- GF(field number, subfield number) or GF(field number)
- EF(field number)

They are used for communicating to a META, INST, or CMND definition the subfield contents of a statement that calls the defined META, INST, or CMND. For example, consider the META definition:

```
-    CBA       META
-              DATA      AF(2),AF(0)
-    LF(0)     DATA      AF(1)
-              MEND
```

The calling statement

```
    X,3 CBA 7,HOL(BETA),16
```

would generate, when processed, the commands

```
    DATA 16,7
```

from the second line, since argument subfield 2 contains 16, and argument subfield 0 contains 7, and

```
    X     DATA HOL(BETA)
```

from the third line, since label subfield 0 contains X and argument subfield one contains HOL(BETA). Note that counting starts at 0.

```
    CBA    META
           DATA   GF(2,2),GF(2)
    GF(0)  DATA   GF(2,1)
           MEND
```

The above META definition is equivalent to the META defined on the previous page.

Since LF, CF, AF and, in general, GF differ only in what subfield they refer to, the following discussion will be generalized to apply to all fields. The general format of a statement calling a META or an instruction could be depicted as:

Label field	Command field	Argument field	3rd field
GF(0,0),GF(0,1)	GF(1,0),GF(1,1)	GF(2,0),GF(2,1)	GF(3,0),GF(3,1)

or,

LF(0),LF(1)	CF(0),CF(1)	AF(0),AF(1)

where G(i,j) represents the jth subfield for the ith field, and LF(i),CF(i),AF(i) represent the label, command and argument subfields, respectively. Note that the first subfield of a field has a zero subscript. If GF is used to refer to the first subfield of a field, the subfield index may be omitted. For example, GF(2) instead of GF(2,0). The META or instruction definition can refer to, and thus operate upon, any of these fields and subfields, by placing the proper field function (LF,CF,AF, or GF) before parentheses containing the appropriate subscript or subscripts. Certain distinctions between conventional macro or meta assemblers and DUAL in regard to this section should be pointed out.

Unlike conventional macro assemblers, the arguments (used in the general sense) that are part of the META do not have to be listed in the META definition statement (the first statement in the META's definition; CBA META in the previous example). With macro assemblers the arguments usually are listed in the macro statement and they frequently are required to appear in the same order as within the body of the macro. DUAL's method results in both coding simplicity and greater flexibility. It should be noted that the field designator functions cannot be used recursively. Unlike conventional meta assemblers, in DUAL the calling statement's subfields may be utilized in both a textual and a symbolic manner. For example, a META may be defined that contains two statements such as

DATA AF(0) and DATA HOL(AF(0))

Assume for a particular call of the META, AF(0) is ALPHA. The first statement would result in DATA ALPHA where ALPHA is interpreted as a symbol. The second statement would generate DATA HOL(ALPHA) where ALPHA represents a textual string of characters.

The expanded field function allows a user to reference an entire field from within a META (as opposed to a subfield). For example:

```
DATA      AF(0),AF(1),AF(2).......AF(n)
```

may now be coded as:

```
DATA      EF(2)
```

The argument to the expanded field function must be an evaluatable expression that results in a non-negative parameter.

It should be noted that the field functions along with the character manipulation functions are used within META's to perform character substitution. It is illegal to use these functions as argument to other character substitution functions.

5.3 ASTERISK FIELD DESIGNATORS (AFA,CFA,GFA)

These functions are known as Argument Field Asterisk (AFA), Command Field Asterisk (CFA), and General Field Asterisk (GFA). These functions have the following general formats:

- AFA(subfield number)
- CFA(subfield number)
- GFA(field number, subfield number)

They are used for communicating to a META, CMND, or STRUCTURE-INST definition, a particular characteristic of the referenced subfield. The characteristic is, whether the first character of the referenced subfield is an asterisk. If it is, the reference results in a value of one, otherwise it results in a value of zero. For example, consider the META definition:

```
F       META
K       EQU   AFA(0)
        MEND
```

The call F *CG would generate K EQU 1, (since the referenced subfield begins with *). Whereas, the call F CG would generate K EQU 0.

The AFA and CFA functions are primarily used with the STRUCTURE-INST directives to define the method of handling indirect addressing which traditionally is represented by a leading *. The subscript(s) within the parentheses following the AFA, CFA, or GFA indicates the subfield to be checked for an asterisk. The subfield counting begins with subscript zero as with LF,CF,AF,and GF.

5.4 NUMBER OF FIELDS OR SUBFIELDS (NUM)

The NUM function provides a technique for informing the META definition of the number of subfields within a field of a call or the number of fields on a calling statement. There are two valid NUM formats:

- NUM(GF) provides the number of fields on the calling statement.
- NUM(i) provides the number of subfields in the ith field.

For example, if a META definition includes the statement:

```
DATA NUM(2)
```

and the META is called by

```
A,B,C      REST,5     9,8,7,6    7,8,9
```

the result will be

```
DATA       4
```

since there are four subfields in the argument field of the call. Similarly, NUM(0) and NUM(1) would generate 3 and 2 respectively.

5.5 CHARACTER COUNT (BY)

Within a META the user can access the number of characters in a field of the META call line by enclosing the field number in parentheses and preceding the parentheses with the symbol BY. BY(1) indicates the number of characters in the command field. In conjunction with the function CH, this permits the user to perform his own scan of the input fields, if he desires. The symbol BY can only be used within a META.

5.6 CHARACTER COUNT (BY#)

The BY# function is identical to the BY function except that it treats the special characters designated by a quote (´) as a single character (see BY description above). For example, assuming a META call line argument field is:

```
HIHO+77/66,55,ABCDE´L´R
then, BY(2) IS 23
while, BY#(2) is 21
```

5.7 CHARACTER SUBSCRIPT (CH)

A character subscript is coded as three expressions (the last one is optional), separated by commas, enclosed in parentheses and preceded by the term CH. The first expression specifies which field, the second expression specifies the initial byte, and the third expression indicates the number of characters. The CH function allows the user to access a character string from the specified field of the calling line of a META. If the number of characters is missing, it is assumed to be one.

For example, assume that line 1 is the calling line of a META and line 2 lies within a META

```
-    MATRIX INVERTER A,B,C
-    JUMPSYM CH(2,0,6),EQ,INVERT,$2
```

then line 2 would check to see whether CH(2,0,6), that is, characters 0 through 5 of the argument field, of the calling line are "INVERT".

5.8 CHARACTER SUBSTITUTION (C#)

The C# function is identical to the CH function except that it treats the special character designated by a quote (´) as a single character (see CH description on page 5-5). For example, assuming a META call line argument field is:

```
5+´LXY´R,"ABC,77"
then, CH(2,1,8) = +´LXY´R,
while, C#(2,1,8) = +´LXY´R"A
```

5.9 SYMBOLIC CONCATENATION (SC)

Symbolic concatenation allows dynamic construction of symbols at processing time. If referenced, the constructed symbol must be defined elsewhere in the program; otherwise an undefined symbol indication will be generated. Symbolic concatenation cannot be used recursively (cannot call itself) nor can it be used in combination with numeric concatenation.

The format for usage of symbolic concatenation is SC(a,b,.....) where "a" and "b" represent symbolic or numeric strings of characters. Either "a" or "b" can be a field function (AF,CF,LF,GF). Symbolic concatenation is valid only within a META. For example, assume the calling line of a META is

```
        APPLY                    10,VOLTS,AIRCRAFT$HEADING
```

and that the following input statement is within the APPLY META:

```
        CALL                     SC(APPLY$,AF(1))
```

then this line would generate the following when processed using the above META call line.

```
        CALL                     APPLY$VOLTS
```

5.10 NUMERIC CONCATENATION (NC)

Numeric concatenation provides the user a mechanism for dynamically creating unique symbols. This feature allows the dynamic construction of symbol strings along with the concatenation of the decimal value of the internal parameter, XNC.

If referenced, the constructed symbol must be defined elsewhere in the program, otherwise an undefined error message will be generated. Numeric concatenation cannot be used recursively (cannot call itself) or in combination with symbolic concatenation.

The format for using numeric concatenation is either NC(a,b) or NC(a) where "a" represents a symbolic string and "b" represents a symbolic or numeric string of characters. Either "a" or "b" can be a field function (LF,CF,AF,GF). Numeric concatenation is only valid within a META.

Assuming that the current value of XNC is equal to 5, then

NC(DUAL) would generate DUAL5.

NC(UNIQUE,AF(2)) where AF(2)) equals SYMBOLS
would generate UNIQUESYMBOLS.

5.11 SYMBOLIC VALUE (SV)

The user can concatenate the symbolic value of a symbol by enclosing the symbol in parentheses and preceding it by SV. For example:

```
OMEGA   EQU  HOL(XYZ)
XXX     EQU SC(ALPHA,SV(OMEGA))
```

Assuming this line is in a META, then this will generate the new character string ALPHAXYZ. It should be noted that the SV function will not work if a PROHOL statement is active.

5.12 VALUE STRING (VS)

The VS function is used in conjunction with the string directive. The VS function retrieves the characters in a string definition. The VS function can only be used within a META. For example, assume the following string definitions:

```
GOOD          STRING         GOOD
BAD           STRING         BAD
BLANK         STRING
OR            STRING         OR
```

Then the following line within a META

```
DATA          HOL(VS(GOOD)VS(BLANK)VS(OR)VS(BLANK)VS(BAD))
```

would generate

```
DATA          HOL(GOOD OR BAD)
```

5.13 INDIRECT VALUE STRING (IS)

The IS function is used in conjunction with the STRING directive. It is used when a STRING variable contains the name of a string which has the desired character string. The IS function can only be used within a META. The following example will help illustrate its use:

```
STRINGPTR          STRING     XSTRING
XYZ                META
XSTRING            STRING     GF(2)
                   DATA       HOL(IS(STRINGPTR))
                   MEND
                   XYZ        AOK
```

would generate

```
              DATA       HOL(AOK)
```

5.14 NUMERIC (N#)

The N# function is a substitution function and can only be used within a META. The argument must be a legal expression resulting in a non-negative parameter. The result is then changed into the corresponding numeric character representation. For example:

```
SQR                META       3
#NUM               SET        AF(0)*AF(0)
                   DATA       N#(#NUM)
                   MEND
```

then

```
                   SQR        20 would generate
                   DATA       400
```

5.15 SPECIAL CHARACTER (SP)

The SP function is a substitution function and can only be used within a META. The argument can be any single character including blanks. The SP function is used to generate special characters that are normally illegal in DUAL syntax, including the . , blanks,; etc.

5.16 Character Value (CV)

The character value function allows the user to obtain the value of a character in EBCDIC. The CV function can only be used in a META. For example,

```
Label           Command         Argument
field            field           field

SQUEEZE         META
I               LOOP            0,1,BY(2)-1
                JUMPVAL         CV(2,I),EQ,C( ),$TEST
                DATA            CV(2,I)
                LOOPTEST
                MEND
```

Note that the first argument specifies the field, while the second argument specifies the character index.

5.17 SYMBOL TYPE (TYPE)

The TYPE function permits the user to access the classification value assigned a symbol by DUAL's symbol classification scheme. The symbol will be processed as command or non-command based on the DUALPROC directive. Non-command symbols are symbols whose types are 9 or less. Normally it is better to utilize the EXP function if types 0-7 of an expression are being evaluated.

The symbol classifications and corresponding values are:

0	parameter
1	address
2	forward reference (undefined)
3	external reference
4	list
5	machine list
7	character string
8	structure
9	function
10	directive
11	meta
16-24	built in instruction
25-63	instructions defined via STRUCTURE

For example:

```
    S              DATA                 TYPE(NUM)
```

would generate a 9 at location S

```
    P              EQU                  TYPE(DATA)
```

would set P equal to 10.

5.18 EXPRESSION TYPE (EXP)

The EXP function permits the user to determine the type of expression being scanned. The various types are identical with those discussed for the TYPE function.

For example, assume the following three symbols X, R, and P are an external, a relocatable address and a parameter. Further, assume that line 1 below represents a call to the META DECLARE and line 2-4 lies within the definition of the DECLARE META.

```
        (1)                  DECLARE X,R,P

        (2)$1                EQU EXP(AF(0))
        (3)$2                EQU EXP(AF(1))
        (4)$3                EQU EXP(AF(2))
```

Then lines 2-4 would set $1 EQU 3(AF(0))=X which is an external), $2 EQU 1 (AF(1))=R which is a relocatable address) and $3 EQU 0(AF(2))=P which is a parameter.

5.19 REMAINDER (REM)

The REM function allows a user to access the remainder when an address expression or value is divided by a specified amount. The format for REM is REM(Exp,Divisor). The REM function supplies the remainder of a division. The REM function initial argument must be a parameter or an address (defined prior to usage), and the final arguments must be a parameter.

For example: JUMPVAL REM(DELTA,2),EQ,0,$3 would cause a skip to $3 if the address of DELTA fell on an even boundary.

The usage REM(P,13) supplies a value of 4 when P is equal to 30.

5.20 PARAMETER (PAR)

An expression can be classified as a parameter by enclosing the expression in parentheses and preceding it by the term PAR. This is not necessary if the expression would automatically be classified as a parameter.

5.21 ABSOLUTE VALUE (ABS)

The user can obtain the absolute value of an expression by enclosing the expression in parentheses and preceding it by ABS. For example, assuming ALPHA is -10, ABS(ALPHA*5) would generate a positive 50.

5.22 BIT MASK (BIT)

The user can generate a mask for any bit by enclosing the desired bit position within parentheses preceded by the term BIT. Bits are numbered from left to right starting at 0 (where 0 is usually a sign bit). For example:

```
BIT(2)
```

is the same as octal 1000 on a computer with a 12 bit word size.

5.23 RIGHT JUSTIFIED MASK (MASK)

The user can generate a mask that is right justified by specifying the number of bits desired enclosed in parentheses and preceded by the term MASK. The number of bits must be between 1 and the word size of the target machine. For example:

```
DATA MASK (6) is the same as DATA HEX (3F).
```

5.24 LEFT JUSTIFIED MASK (LMASK)

The user can generate a mask that is left justified by specifying the number of bits desired enclosed in parentheses and preceded by the term LMASK. The number of bits must be between 1 and the word size of the target machine. For example,

```
DATA  LMASK(5+5)
```

is the same as DATA HEX(FFC0) on a 16 bit target.

5.25 ADDRESS (REL)

An expression can be classified as an address (with the current location counters section number) by enclosing it in parentheses and preceding it by REL.

- REL(100)

5.26 ADDRESS CLASSIFICATION (ADR)

DUAL provides a function for classifying an address into one of eight categories. The eight different categories with their corresponding values are:

0 = Absolute
1 = Absolute,Index
2 = Indirect Absolute
3 = Indirect Absolute,Index
4 = Address
5 = Address,Index
6 = Indirect Address
7 = Indirect Address,Index

The format for using the address classification function is shown below. Note thate the indirect indicator is the presence or absence of an asterisk prior to the address:

```
ADR (Address,Index)
```

Assume that A, R, and X represent an ABSOLUTE, RELOCATABLE and INDEX respectively. Then the following examples will help illustrate the usage of the ADR function.

```
CLASS          META
B              EQU        ADR(AF(0),AF(1))
               MEND
```

The CLASS META sets item B to the "type of address" where AF(0) is the address and AF(1) is the index.

Label field	Command field	Argument field
	CLASS	A

Would set B = 0

	CLASS	*R,X

Would set B = 7

	CLASS	*R

Would set B = 6

	CLASS	A,X

Would set B = 1

5.27 ADDRESS MODIFICATION (CA,HA,WA,DA)

Four functions are provided in DUAL for address modification. Address modification may be used to modify any address or to specify the addressing mode during instruction definition (STRUCTURE or CMND directive). The address modification functions will not produce meaningful results when used recursively. The functions are:

- CA = *character address*
- HA = halfword address
- WA = word address
- DA = doubleword address

Example usage:

```
        DATA          CA(ALPHA)
```

Assuming ALPHA is equal to relocatable word address 0100, then CA(ALPHA) would be defined as relocatable character address 0400 in a machine with 4 characters per word.

Note that the default case for all addresses is word relocation.

5.28 ADDRESSING (BR,DISP)

DUAL provides two functions that aid in calculating addresses as displacements; BR and DISP. These two functions in conjunction with the directives BASE and RELEASE, allow the user to shift the burden of addressing in base register computers from the programmer to the DUAL processor. DUAL must operate in the TWO/PASS mode for these functions to operate properly.

The BASE directive informs DUAL which registers are to be designed as base registers and what the contents are to be. The RELEASE directive informs DUAL which registers are no longer to be considered as base registers.

The format for BR (Base Register) and DISP (DISPlacement) are:

```
BR        (Base register subfield, address subfield)
DISP      (Base register subfield, address subfield)
```

The base register subfield must contain either a base register designation (an expression which results in a value between 1 and 15) or be void. The user may designate which base register he desires. In that event, he would put a displacement from his specified base register in the address subfield. If the user desires DUAL to calculate the base register and displacement, he merely writes a symbolic address in the address subfield and leaves the base register subfield void.

The following examples will help illustrate the base register concept:

Assume a computer instruction format as shown below:

```
0........7    8...11     12...15   16....19     20.......31
Operation     General    Index      Base        Displacement
   Code       Register             Register
```

A typical instruction that utilizes base registers could be defined as follows:

Label field	Command field	Argument field
	SIZE	32,8,4096

Defines the target machine's word size as 32, address size as 8 and maximum displacement as 4096.

L1	LIST	8,4,4,4,12

Defines bit pattern for instruction

LOAD	CMND,L1	7,CF(1),AF(1),BR(AF(2), AF(0)),DISP(AF(2),AF(0))

This defines a "LOAD" instruction designating that addresses are to be calculated as displacements from a base.

	BASE	14,BEGIN

Tells DUAL that base register 14 is an active base register with a value equal to BEGIN.

BEGIN	LOAD,14	$
	BRANCH	PGMSTART

This instruction actually loads register 14 so that it may be used as a base register. Note that it is the programmer's responsibility to maintain the base registers

BVAL	DATA	$+4095

Data word for loading a subsequent base register.

Label field	Command field	Argument field
	LOAD,5	BVAL
	LOAD,5	8,,14

The above two statements are identical and demonstrate the two alternative formats for using the LOAD instruction. In the first one, the displacement is calculated as BVAL-BEGIN (which is 8 units, assuming each instruction is 32 bits). The second example demonstrates how the user can supply the actual value if he so desires. The two alternative formats are:

```
LOAD,REGISTER  SYMBOLIC ADDRESS,INDEX
LOAD,REGISTER  DISPLACEMENT,INDEX,BASE REGISTER
```

5.29 SECTION NUMBER (SN)

The user can access the section number for an address by enclosing the address in parentheses and preceding it by SN. The default section number is one, which is synonymous with the RELOCATE or blank CSECT directives. The ABSOLUTE directive sets the section number to 0. Subsequent named CSECTs, DSECTs, and COMMONs set the section number to two, three...up to thirty-one. DUAL allows a maximum of 32 distinct sections per module.

- SN(ALPHA)

5.30 TEXTUAL CHARACTER ADDRESS (CHAR)

The user can obtain the character address of a textual constant by enclosing the textual address in parentheses and preceding it by CHAR. The CHAR function is used in conjunction with the DATUM, ADJUST, and RESERVUM directives.

- CHAR(OMEGA)

5.31 TEXTUAL ADDRESS OFFSET (OFF)

The user can obtain the character offset of a textual address by enclosing the textual address in parentheses and preceding it by OFF. The OFF function is used in conjunction with the DATUM, ADJUST, and RESERVUM directives.

5.32 ABSOLUTE ADDRESS (ADDR)

The user can specify an absolute address by enclosing in parentheses the section number and offset separated by a comma and preceded by the term ADDR.

- ADDR(5,100)
- ADDR(SN($),2000)

The latter example would use the current section number since the $ symbol always contains the current or active section number.

5.33 PAGE ADDRESS (PAGE)

An address may be calculated as a page address (PAGE bits only) by enclosing an expression in parentheses and preceding it by the term PAGE. The PAGE function works in conjunction with the SIZE directive and is only meaningful for the DATA directive and certain built-in target instructions.

For example: Assume ALPHA is at location 3512 hexadecimal, then if the following statements appear in a program:

```
SIZE      16,8,256
 .
 .
 .
DATA      PAGE(ALPHA)
```

with a load bias of 240 hexadecimal, the load module output of the linkage editor would be 3700 hexadecimal.

5.34 PAGE PLUS DISPLACEMENT (PAGCON)

An address may be calculated as a page address along with a displacement by enclosing two expressions separated by commas enclosed in parentheses and preceded by the term PAGCON. The PAGCON function works in conjunction with the SIZE directive and is only meaningful for the DATA directive and certain built-in target instructions.

For example: Assume ALPHA is at location 3512 hexadecimal, then if the following statements appear in a program:

```
        SIZE      16,8,256
        DATA      PAGCON(ALPHA,77)
```

with a load bias of 240 hexadecimal, the load module output of the linkage editor would be 374D hexadecimal.

5.35 LOWER BIT RELOCATION FACTOR (LB)

The user can obtain the lower half-word relocation of an address by enclosing the address in parentheses and preceding it by LB. For example, assume a 16 bit computer and that ALPHA is assembled in section one with an offset of hex 250. Further, assume that the relocation factor for the current modules Section 1 is 220. Then: LB(ALPHA) would generate a 70. It should be noted that the LB function is only meaningful with the DATA and GEN directives and certain built-in target instructions.

5.36 HIGH BIT RELOCATION FACTOR (HB)

The user can obtain the upper halfword relocation of an address by enclosing the address in parentheses and preceding it by HB. For example, assume a 16 bit computer and that ALPHA is assembled in section one with an offset of hex 250. Further assume the relocation factor for the current module section one is 220. Then: HB(ALPHA) would generate a 4. It should be noted that the HB function is only meaningful for the DATA and GEN directives and certain built-in target instructions.

5.37 NUMBER OF ELEMENTS IN LIST (NL)

The number of elements in a list may be referenced by enclosing the name of the list in parentheses and preceding it by NL. For example, assume a list definition like line 1. Then line 2 would set ELEMENTS equal to 6.

```
        STRING          LIST            10,5*ABC,4,16,HEX(ABC),7
        ELEMENTS        EQU             NL(STRING)
```

5.38 EXPANDED LIST (EL)

The expanded list function allows a user to expand a list to its current set of values. The EL function works in conjunction with the LISTMODE directive. For example:

```
        X               LIST            5,4,3
        Y               LIST            5,EL(X)
        Z               LIST            EL(X),EL(Y),5,(EL(X))
```

would define Z as a list of

```
        5,4,3,5,5,4,3,5,(5,4,3)
```

The argument for EL must be the name of a list. If EL is used within a META then the argument can designate a particular subfield where a list name appears, for example:

```
                        XXX             5,X,10
                        .
                        .
                        .
        XXX             META
        L1              LIST            AF(2,1),5         ASSUMING X IS
                                                          A LIST NAME
```

5.39 DEPTH OF A LIST (DEP)

the depth of a list is the number of parentheses surrounding a particular list element plus one. If the element is nonexistent, then the depth is zero. For example, using the following list:

```
        X               LIST            DIVE,(TO,(VERY GREAT),
                                        AND),FAR,((DEPTHS))
```

then

```
DEP(X)=1
DEP(X,0)=1
DEP(X,1)=2
DEP(X,2)=1
DEP(X,3)=3
DEP(X,1,0)=1
DEP(X,1,1)=2
DEP(X,3,0)=2
DEP(X,1,2)=1
DEP(X,1,3)=0
```

5.40 LOCAL SYMBOL VALUE (LSV)

The user can obtain the value of a local symbol at a previous META local nesting level by enclosing a local symbol in parentheses and preceding it by LSV. DUAL will search sequentially previous nesting levels for the desired local symbol. For example:

```
XX          META        1
$5          EQU         1
            YY
            MEND

YY          META
$5          EQU         2
            DATA        $5
            DATA        LSV($5)
            MEND
            XX
```

In the above example two data cells would be generated. The DATA $5 would generate a 2, while the DATA LSV($5) would generate a 1.

5.41 USER VALUE (UV)

The function UV is an abbreviation for the term "user value". This function in conjunction with the VALUE directive permits the user to assign values to a symbol in addition to the value that the DUAL processor has assigned to that symbol. For example, if the symbol QUAKE has the location address 523 and has been assigned the user value 17 then:

```
S2          DATA QUAKE puts 523 in S2.
```

whereas

```
S2          DATA UV(QUAKE) puts 17 in S2.
```

A user value is assigned by using the VALUE directive. Thereafter, the user value may be utilized by enclosing the symbol in parentheses and preceding it by UV. If multiple user values are assigned to a symbol, then they can be accessed by enclosing the symbol in parentheses followed by a comma and an index. UV(SYMBOL,INDEX).

The symbols will be processed as command or non-command based on the DUALPROC directive.

The UV function is an extremely powerful tool in developing a META language. The user can assign values to a symbol and later make decisions based on these values as to what type of code to generate.

5.42 OPERATION CODE (OP)

This function is used for defining the operation code value for instruction definitions. OP is used only in conjunction with the STRUCTURE directive.

5.43 DUALPASS

The user can determine which pass DUAL is in by utilizing the system parameter DUALPASS. It is the user's responsiblity in using DUALPASS not to generate more object in one pass than another.

5.44 DUALDATE

The user can access the current date of assembly by utilizing the system parameter DUALDATE (if the host computer's operating system facilitates this function). The format of this parameter is operating system dependent.

5.45 DUALTIME

The user can access the time of assembly by utilizing the system parameter DUALTIME (if the host computer's operating system facilitates this function). The format of this parameter is operating system dependent.

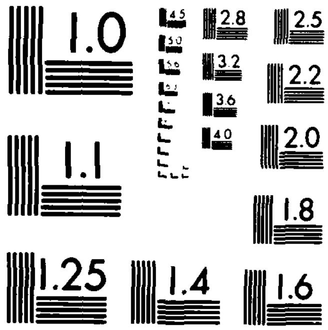
5.46 LINE

The user can access the current listing line number by utilizing the system parameter LINE. For example:

```
- DATA LINE
```

AD-A150 584 PROCEEDINGS OF THE TECHNICAL FORUM (3RD) ON THE F-16 MIL-STD-1750A MICROP..(U) AERONAUTICAL SYSTEMS DIV WRIGHT-PATTERSON AFB OH J L PESLER ET AL. 06 MAY 82 5/6

UNCLASSIFIED ASD-TR-82-5011-VOL-2 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

CHAPTER 6

DIRECTIVES

Commands in DUAL are termed directives. Just as machine instructions are used to request the computer to perform a sequence of operations, directives are used to instruct DUAL to perform specific operations at translation time. Directives are used in formatting both input and output data, in defining data, controlling the sequence of input statement processing, generating higher order languages and defining environmental conditions. DUAL's directives have been separated into thirteen functional categories.

Examples are provided with each directive which depict the usage of the directive and the required input statement format. Since DUAL is essentially "free-form", card columns are not indicated in the examples. Instead, each directive's format is described on a field basis. A void field means that the directive does not process that field (or subfield) and information in those fields will be ignored. If the label field is described as "not used", the label field can still be used as the goal of a skip to statement (skip to statements alter the sequence of input statement processing).

6.1 Input Control Directives

6.1.1 General

The input control directives are used to define the format and manner in which input statements are to be processed. Via the input control directives the user has the capability to define input formats to suit his data processing requirements. The input control directives do not generate any object code. The user is also provided an update feature for maintaining his source program. Unlike other language processors, DUAL does not require a special pass for updating.

6.1.2 COLUMN

6.1.2.1 Format

Label field	Command field	Argument field
	COLUMN	Exp,Exp,Exp

6.1.2.2 Where

Label field -- not used
Command field -- COLUMN
Argument Field -- The argument field must contain three subfields, each of which contains an evaluatable expression. The contents of the subfields are:

AF(0) = Last meaningful column on input statement. The column immediately following this is used to indicate that the input statement consists of more than one line (continuation).

AF(1) = Continuation column. This is the initial column to be used for continuation lines.

AF(2) = Initial column. This is the column which DUAL will initially process. It is the beginning of the label field.

6.1.2.3 Function

The COLUMN directive is used to define the last, continuation and initial columns on subsequent symbolic input statements. The columns prior to the initial column and after the column used to designate continuation are not processed by DUAL and can be used for any purpose the user desires (e.g. sequence numbers).

6.1.2.4 Error Conditions

- Last meaningful column not between 41 and 76.
- Continuation or initial column not between 1 and 40.

6.1.2.5 Example Usage

Label field	Command field	Argument field
	COLUMN	70,16,1

This statement would define the initial column = 1, the continuation column = 16, and the last meaningful column as 70. Continuation would be denoted in column 71.

	COLUMN	72,45,1

This statement would result in an error since the continuation column exceeds 40.

6.1.3 LENGTH

6.1.3.1 Format

Label field	Command field	Argument field
	LENGTH	Exp,Exp

6.1.3.2 Where

Label field - not used
Command field - LENGTH
Argument field - The argument field must contain either one or two evaluatable expressions.

AF(0) = Legal separation distance between fields.

AF(1) = Maximum fields for processing.

6.1.3.3 Function

The LENGTH directive is used to define the maximum legal separation distance (number of blanks) between fields, and the maximum number of fields to be processed per source statement.

6.1.3.4 Error Conditions

- Illegal length specification (must be void or between 10 and 60)
- Illegal maximum field specification (must be void or greater than 2)

6.1.3.5 Example Usage

Label field	Command field	Argument field
	LENGTH	20,100

Defines the legal separation distance between fields as 20, and tells DUAL to process up to 100 fields for each statement.

6.1.4 LIBRARY

6.1.4.1 Format

Label field	Command field	Argument field
	LIBRARY,print indicator	Symbol(1),Symbol(2) ...Symbol(N)

6.1.4.2 Where

Label field -- not used
Command field -- The initial command field, CF(0), must contain the symbol LIBRARY. CF(1) is used to indicate whether statements processed from the LIBRARY device are to be listed (1 = print, 0 or void = no print).
Argument field -- The argument field must contain one or more subfields each of which contains a legal symbol.

6.1.4.3 Function

DUAL will accept source input statement from three distinct logical devices: the source input device, the update input device and the library input device. The LIBRARY directive will cause DUAL to process subsequent source input statements from the library device. This directive is used in conjunction with the LEND (LIBRARY END) directive. DUAL will search the library file serially until it finds the designated symbol (from the LIBRARY argument field) in the label field of a LIBRARY input statement. It will begin processing source lines from the library device until a LEND directive is processed. Multiple Library requests must be sequential. After the last library request is processed, the library device will be rewound, if applicable.

6.1.4.4 Default Case

DUAL is initialized to process input statements from the source input device.

6.1.4.5 Error Conditions

- Illegal symbol in argument field
- Symbol not found in label field of library

6.1.4.6 Example Usage

Label field	Command field	Argument field
	LIBRARY	SYSTEM1

This would cause DUAL to search the library for the character string "SYSTEM1" in the label field of a LIBRARY input statement. When found, DUAL would begin processing subsequent lines from the library input device.

	LIBRARY	123

This input statement would result in an error since "123" is not a legal symbol.

6.1.4.7 Special Comments

The library device can be used for a variety of purposes. Instruction sets defining different target computers, higher order languages defined via METAs or system parameters, might be contained on a library. The user can make more than one library call. The library is searched sequentially for a single library request, thus multiple arguments implies the user is knowledgeable about the order of information in the library.

6.1.5 LEND

6.1.5.1 Format

Label field	Command field	Argument field
	LEND	

6.1.5.2 Where

Label field -- not used
Command field -- LEND
Argument field -- not used

6.1.5.3 Function

The LEND directive signifies the end of the library mode (see LIBRARY directive). Subsequent source inputs will be processed from the previous input device.

6.1.5.4 Example Usage

Label field	Command field	Argument field
	LEND	

Signifies end of library mode.

6.1.5.5 Special Comments

The library device can contain more than one LEND input statement since the LIBRARY directive will search for the proper starting point within the library device.

6.1.6 ALTER

6.1.6.1 Format

Label field	Command field	Argument field
.	ALTER	Expression Expression(opt.)

6.1.6.2 Where

Label field -- The initial character must be a period.
Command field -- ALTER
Argument field -- The initial argument subfield, AF(0), must contain an evaluatable expression which results in a positive value that is greater than any value in any previous ALTER statement's argument subfields. The second argument subfield, AF(1), may contain an evaluatable expression which resolves to a value greater than or equal to the initial argument subfield, or it may be void.

6.1.6.3 Function

The ALTER directive enables the user to update his source program while it is being processed. The function of an ALTER statement depends on whether there are one or two expressions in the argument field. A single expression in the argument field of an ALTER card enables the user to insert statements from the update device prior to (before) processing the specified line from the source device. When an ALTER statement contains two expressions in the argument field, it acts as a change function. That is, it will insert statements prior to the initially specified statement from the source device and will then skip lines until it is beyond the final specified line from the source device. There is no requirement for a one-to-one correspondence of source and alter statements when using the ALTER directive.

6.1.6.4 Error Conditions

- Not in UPDATE MODE
- Illegal expression
- Label field does not contain a period as its initial character
- Alter statements not in numeric order

6.1.6.5 Example Usage

Label field	Command field	Argument field
.	ALTER	175
	DATA	5
	MOVE	ALPHA,BETA

Would insert above two cards prior to input statement 175.

.	ALTER	180,180
	CLA	BETA
	STO	GAMMA

Would replace input statement 180 with the above two lines.

.	ALTER	300,654

Would delete input statement 300 through 654.

.	ALTER	70

Illegal since 70 is less than 654; updates must be in order.

6.1.6.6 Special Comments

When DUAL is in the update mode, it will automatically output a new update source on the update output device. The update sequence numbers will appear on the listing output. Update inputs will, in addition, contain an * on the listing output. DUAL does not require a special pass in order to update a source program.

6.1.7 ALTEND

6.1.7.1 Format

Label field	Command field	Argument field
.	ALTEND	

6.1.7.2 Where

Label field -- The initial character must be a period.
Command field -- ALTEND
Argument field -- Not used

6.1.7.3 Function

Terminating the updating of the source file.

6.1.7.4 Error Conditions

Label field	Command field	Argument field
.	ALTEND	

Ends update mode.

6.1.8 LIBMODE

6.1.8.1 Format

Label field	Command field	Argument field
	LIBMODE	ON or OFF

6.1.8.2 Where

Label field -- not used
Command field -- LIBMODE
Argument field -- ON or OFF

6.1.8.3 Function

The LIBMODE directive is used to control what portion of the library is processed during a two/pass assembly. Its function is to save assembly time. The default mode is ON, which signifies that subsequent library source will be processed both passes. The OFF designator causes subsequent source to be processed the first pass only.

6.1.8.4 Error Conditions

- Illegal library mode designator

6.1.8.5 Example Usage

Label field	Command field	Argument field
	LIBMODE	OFF
	LIBMODE	ON

6.1.9 SYNTAX

6.1.9.1 Format

Label field	Command field	Argument field
	SYNTAX	Target name

6.1.9.2 Where

Label field -- not used
Command field -- SYNTAX
Argument field -- The argument field specifies a legal DUAL target name. Appendix T describes the legal target names. In addition, the user can specify "DUAL".

6.1.9.3 Function

The SYNTAX directives allows the user to translate programs written in the syntax of the manufacturers assembly language (for those targets designated in Appendix T). Since most "built-in" targets have a unique "DUAL syntax", the user can switch back and forth between the two different syntaxes (DUAL and the traget, e.g., PDP-11) as often as he deems necessary.

6.1.9.4 Default Case

If the user specifies SYNTAX mmmmm then the DUAL system will expect subsequent source lines in the target's syntax. If no SYNTAX is specified (DUAL as a generalized meta assembler), then the syntax is DUAL.

6.1.9.5 Error Conditions

- Illegal target name specification

6.1.9.6 Example Usage

```
    SYNTAX     PDP11
    SYNTAX     DUAL
    SYNTAX     I8080
    SYNTAX     NOVA
```

6.1.10 NUMBASE

6.1.10.1 Format

Label field	Command field	Argument field
	NUMBASE	expression

6.1.10.2 Where

Label field -- not used
Command field -- NUMBASE
Argument field -- an evaluatable expression which results in a value between 2 and 32.

6.1.10.3 Function

The NUMBASE directive sets the base for numbers with leading zeroes. The default zero number base is 8 (octal). The user may reset the zero number base as often as he deems necessary. The characters A-Z are used after the digits 0-9 to represent base constants.

6.1.10.4 Error conditions

- Illegal expression
- Value is not between 2 and 32

6.1.10.5 Examples

```
NUMBASE        16
DATA           0123     would generate hexadecimal 123
NUMBASE        8
DATA           01234567  would generate octal 1234567
NUMBASE        2
DATA           010111   would generate binary 10111
```

6.1.11 ACCEPT

6.1.11.1 Format

Label field	Command field	Argument field
	ACCEPT	

6.1.11.2 Where

Label field -- not used
Command field -- ACCEPT
Argument field -- not used

6.1.11.3 Function

The ACCEPT directive can only be used within a META. It allows the META writer the capability of processing multiple source statements from within a META. The ACCEPT statement causes DUAL to read the next source statement and replace the previous calling META's arguments with the just read source statement.

6.1.11.4 Error Conditions

- ACCEPT directive is not within a META-MEND

6.1.11.5 Example usage

Assume the following META:

```
STUDENTS     META       3                     . This META reads
                                                a set of subsequent
                                                statements containing
                                                the names of all the
                                                students. The origi-
                                                nal call line has the
                                                number of students in
                                                the class.

             JUMPVAL    AF(0),LT,1,$BAD       . Make sure at least
                                                one student.

             JUMPVAL    AF(0),GT,100,$BAD     . Class will be too
                                                crowded.

#            LOOP       #,1,AF(0)             . Loop for number of
                                                students.

             ACCEPT                           . Get next student.
             DATA       HOLC(LF(0))           . Generate student's
                                                first name.

             DATA       HOLC(CF(0))           . Generate student's
                                                last name.

             DATA       #,EF(2)               . Generate student
                                                number and other per-
                                                tinent information.

             LOOPTEST

             MEND
```

Then if a source program looked like the following:

```
                STUDENTS   2
    CURT        JONES      MALE,17,150,75
    KAREEM      JABBAR     MALE,34,240,86
```

it would generate:

```
                DATA       HOLC(CURT)
                DATA       HOLC(JONES)
                DATA       1,MALE,17,150,75
                DATA       HOLC(KAREEM)
                DATA       HOLC(JABBAR)
                DATA       2,MALE,34,240,86
```

6.2 Output Control Directives

6.2.1 General

The output control directives are used to generate user defined object, format the listing output and also to select which portion of the input program is to be listed. Via the output control directives the user can select what title shall appear at the top of each page, how much information shall be contained on a particular page, how many spaces between lines and which lines shall be listed.

6.2.2 PRINT

6.2.2.1 Format

Label field	Command field	Argument field
	PRINT	Expression (optional)

6.2.2.2 Where

Label field -- not used
Command field -- PRINT
Argument field -- The initial argument subfield can either be void or contain an evaluatable expression which results in a value of either 0 or 1. A zero will cause the listing output to be displayed in octal, while a one indicates hexadecimal.

6.2.2.3 Function

The PRINT directive causes subsequent input statements to be listed on the output listing device. The PRINT directive is used in conjunction with the PRINTOFF directive.

6.2.2.4 Default Case

DUAL will initially list all input statements.

6.2.2.5 Error Conditions

-Illegal print indicator

6.2.2.6 Example Usage

Label field	Command field	Argument field
	PRINT	

Turn listing indicator on.

	PRINT	1

Turn listing indicator on and set print mode to hexadecimal.

6.2.3 PRINTOFF

6.2.3.1 Format

Label field	Command field	Argument field
	PRINTOFF	

6.2.3.2 Where

Label field -- not used
Command field -- PRINTOFF
Argument field -- not used

6.2.3.3 Function

The PRINTOFF directive suppresses the listing of subsequent input statements on the listing output device. The PRINTOFF directive is used in conjunction with the PRINT directive.

6.2.3.4 Default Case

DUAL will initially list all input statements.

6.2.3.5 Example Usage

Label field	Command field	Argument field
	PRINTOFF	

Suppress listing output starting at this statement.

	DATA	60

This line will not be listed.

	DATA	20

This line will not be listed.

	PRINT	

This and subsequent lines will be listed.

6.2.3.6 Special Comments

The PRINT and PRINTOFF directives are designed to avoid any needless listing of input statements by the DUAL processor. Examples might be a set or target machine instruction definitions (possibly from the library device) or a series of system META's. It should be noted that any input statement that is found in error will be listed even though DUAL is currently suppressing the listing.

6.2.4 PAGE

6.2.4.1 Format

Label field	Command field	Argument field
	PAGE	

6.2.4.2 Where

Label field -- not used
Command field -- PAGE
Argument field -- not used

6.2.4.3 Function

The PAGE directive causes the listing output device to be restored to the top of the page. If the PAGE directive would normally appear at the top of the listing output, then the PAGE directive is ignored. The PAGE directive is never listed (nor is the sequence number displayed).

6.2.4.4 Example Usage

Label field	Command field	Argument field
	PAGE	

Advance listing to top of next page.

6.2.5 SPACE

6.2.5.1 Format

Label field	Command field	Argument field
	SPACE	Expression

6.2.5.2 Where

Label field -- not used
Command field -- SPACE
Argument field -- The initial argument subfield must be void or contain an evaluatable expression which results in a non-negative value.

6.2.5.3 Function

The SPACE directive allows the user to insert blank lines in the listing output. The SPACE input statement itself is never listed (nor is the sequence number displayed), unless there is an error.

6.2.5.4 Error Conditions

- Illegal expression

6.2.5.5 Example Usage

Label field	Command field	Argument field
	SPACE	5

This line would cause the output of five blank lines on the listing output device.

	SPACE	7-10

This input statement would be flagged as an error since the argument field results is a negative value.

6.2.6 TITLE

6.2.6.1 Format

Label field	Command field	Argument field
	TITLE	HOL(title), HOL(subtitle)

6.2.6.2 Where

Label field -- not used
Command field -- TITLE
Argument field -- AF(0) and AF(1) may contain textual constants or be left blank.

6.2.6.3 Function

The TITLE directive enables the user to specify the title and subtitle to appear at the top of each listing output page until another TITLE statement is processed. The title can be changed, while retaining the subtitle, and vice versa, by entering new information only in the field corresponding to that portion of the heading to be changed. If multiple source lines are required for a TITLE statement, then the lines prior to the last line will not appear at the top of the page.

6.2.6.4 Default Case

DUAL will initially list its version number in the title buffer. The subtitle is initially void.

6.2.6.5 Error Conditions

- Argument field does not contain a textual constant.

6.2.6.6 Example Usage

Label field	Command field	Argument field
	TITLE	HOL(ROUTINE XYZ)

This would cause the character string "ROUTINE XYZ" to appear at the top of each subsequent page. The specified title or subtitle may be a maximum of 108 characters in length.

6.2.7 TABSET

6.2.7.1 Format

Label field	Command field	Argument field
	TABSET	Exp,Exp,Exp

6.2.7.2 Where

Label field -- not used
Command field -- TABSET
Argument field -- The argument field must contain three subfields, each of which contains an evaluatable expression. The contents of the subfields are:

AF(0) = Where to begin printing the label field
AF(1) = Where to begin printing the command field
AF(2) = Where to beins printing the argument field

6.2.7.3 Function

The TABSET directive enables the user to specify how DUAL should print source input statements from within a META, a LOOP or WITHHOLD. The user can specify where to begin displaying each of the first three fields. A maximum of 80 characters will be displayed for a line. If lines are to be output onto the symbolic output device, then TABSET can be used to format the output (see META directive).

6.2.7.4 Error Conditions

- Illegal or non-positive expression
- AF(0) greater than or equal to AF(1)
- AF(1) greater than or equal to AF(2)
- AF(2) greater than 80

6.2.7.5 Example Usage

Label field	Command field	Argument field
	TABSET	1,20,40

This statement would cause the label, command and argument fields to be displayed starting in columns 1, 20 and 40.

	TABSET	1,40,35

Illegal since the argument field cannot be displayed prior to the command field.

6.2.7.6 Special Comments

If the label field extends into the command field, or the command field runs into the argument field, etc., then only that portion of the field will be displayed up to the next field.

6.2.8 BINARY

6.2.8.1 Format

Label field	Command field	Argument field
	BINARY	Exp(1),Exp(2)....Exp(N)

6.2.8.2 Where

Label field -- not used
Command field -- BINARY
Argument field -- The argument field must contain one or more subfields, each of which contains an evaluatable expression.

6.2.8.3 Function

The BINARY directive enables a user to generate his own object (loader text) along with the object normally generated by DUAL. For example, if a computer has memory protection, through the usage of the BINARY directive, the object output could contain the type of protection to be used for various segments of a program (e.g., READ ONLY, READ WRITE, etc.). It should be noted that this is only meaningful if the loader for a particular system recognizes the various user defined object types.

6.2.8.4 Error Conditions

- Illegal expression

6.2.8.5 Example Usage

Label field	Command field	Argument field
	BINARY	5*A,B

Outputs user defined object with an initial value of 5*A and a second value of B.

6.2.9 CHECKSUM

6.2.9.1 Format

Label field	Command field	Argument field
Symbol(opt.)	CHECKSUM	

6.2.9.2 Where

Label field -- The initial label subfield may contain a symbol. If a symbol does appear in the label field, then it is defined as the value of the current location counter.
Command field -- CHECKSUM
Argument field -- not used

6.2.9.3 Function

The CHECKSUM directive causes DUAL to output object that tells the DUAL Link Editor to generate a data word which contains a checksum of all previous data words prior to the last checksum (or the start).

6.2.9.4 Error Condition

- Illegal label

6.2.9.5 Example Usage

Label field	Command field	Argument field
CHKI	CHECKSUM	

flags linker to generate a checksum at CHKI.

6.2.10 PUNCH

6.2.10.1 Format

Label field	Command field	Argument field
	PUNCH	

6.2.10.2 Where

Label field -- not used
Command field -- PUNCH
Argument field -- not used

6.2.10.3 Function

The PUNCH directive, in conjunction with the PEND directive, enables the user to output source lines to the symbolic output device (SO) during the translation process. All lines following the PUNCH statement, regardless of contents, will be output in source format until a PEND statement is reached. Normal statement processing does not take place for these lines; therefore, they are ignored after being output. Source output lines are handled as follows:

1. PUNCH directive outside meta processing.

   In this circumstance, the source output lines are merely copied onto the symbolic output device. Therefore, the line structure is not altered in any way.

2. PUNCH directive within meta processing.

   In this case, normal meta line substitution takes place initially. Then the output line is reconstructed with the COLUMN directive parameters providing the basis for the format. Multiple lines may result from one logical statement due to meta line substitution.

6.2.10.4 Example Usage

Label field	Command field	Argument field
	PUNCH	
ALPHA	DATA	5
	PEND	

The line ALPHA DATA 5 will be output to the symbolic output device (SO) without any line format alterations. The statement will not otherwise be processed. Therefore, no object will be generated.

Label field	Command field	Argument field
PUNMETA	META	
	PUNCH	
LF(0)	DATA	HOL(AF(0))
	PEND	
$1	LOOP	1,1,NUM(2)-1
	PUNCH	
	DATA	AF($I)
	PEND	
	LOOPTEST	
	MEND	

This defines the meta PUNMETA whose function is to output source DATA statements, to the symbolic output device, based on the argument field inputs.

```
                          COLUMN          70,20,1
```

This specifies the structure of source input statements and source output lines from within a meta.

```
          ALPHA          PUNMETA          BETA,5,1000,ALPHA
```

This would cause the following lines to be output to the symbolic output device:

```
          ALPHA          DATA             HOL(BETA)
                         DATA             5
                         DATA             1000
                         DATA             ALPHA
```

No object would be generated and the line structure would be based on:

1. initial column = 1
2. last meaningful column = 70
3. continuation column = 20

6.2.10.5 Special Comments

Source output lines will be punched only during the initial pass of the DUAL processing. The source output lines will be ignored during the second pass processing.

6.2.11 PEND

6.2.11.1 Format

Label field	Command field	Argument field
	PEND	

6.2.11.2 Where

Label field -- not used
Command field -- PEND
Argument field -- not used

6.2.11.3 Function

The PEND directive signifies the end of source output processing (see PUNCH directive).

6.2.11.4 Example Usage

Label field	Command field	Argument field
	PEND	

Signifies the end of source output processing

6.2.11.5 Special Comments

If the PEND directive is encountered and source output processing was not in effect (no proceding PUNCH directive), then it will be ignored.

6.2.12 RANGE

6.2.12.1 Format

Label field	Command field	Argument field
	RANGE	Address, Address

6.2.12.2 Where

Label field -- not used
Command field -- RANGE
Argument field -- used to specify range of addresses, where

AF(0) = initial address
AF(1) = final address

6.2.12.3 Function

The RANGE directive is used to specify a range of addresses to be included in the range summary. The user may specify as many ranges (via multiple RANGE statements) as he desires. The range printout shows what lines reference a range of addresses.

6.2.12.4 Error Conditions

- illegal address
- AF(1) address less than AF(0) address
- AF(0) address is not in the same section as AF(1)

6.2.12.5 Example Usage

Label field	Command field	Argument field
	RANGE	ALPHA,BETA

Assuming both ALPHA and BETA are addresses in the same section, then DUAL will include a range printout by line number (unless BYLOC is specified on .DUAL) of all statements that have an address expression which are between ALPHA and BETA inclusively.

6.2.13 LINES

6.2.13.1 Format

Label field	Command field	Argument field
	LINES	EXPRESSION

6.2.13.2 Where

Label field -- not used
Command field -- LINES
Argument field -- an evaluatable expression which results in a positive parametric value.

6.2.13.3 Function

The lines directive allows the user to control the number of lines to be printed per page by DUAL. The default value is 50.

6.2.13.4 Example Usage

	LINES	60

6.2.14 UNDEFINE

6.2.14.1 Format

Label field	Command field	Argument field
	UNDEFINE	EXP,EXP,EXP,EXP

6.2.14.2 Where

Label field -- not used
Command field -- UNDEFINE
Argument field -- The first four subfields must contain evaluatable expressions or be void. The value in AF(0) must be a 0 (generate single ZERO value), a 1 (generate value specified in AF(2), or a 2 (generate values specified in AF(2) and AF(3). AF(1) is the number of bits to generate per undefine value; the default is word size. AF(2) and AF(3) are the initial and final values to be generated.

6.2.14.3 Function

The UNDEFINE directive allows the user to control what values are to be generated for an undefine command.

6.2.14.4 Examples

	UNDEFINE	1,16,HEX(FACE)
	UNDEFINE	2,12,HEX(ACE),07777

6.2.15 HEADING

6.2.15.1 Format

Label field	Command field	Argument field
	HEADING	HOL(character string)

6.2.15.2 Where

Label field -- not used
Command field -- HEADING
Argument field -- The initial argument subfield must contain a textual character string.

6.2.15.3 Function

The HEADING directive allows the user to override DUAL's heading printed just after the title and subtitle lines (LINE... LOCATION... ETC.) and just prior to normal assembly listing lines. The HEADING directive is used in conjunction with the LISTING directive. The HEADING should not be longer than the listing line length defined by the LISTING directive (default maximum length = 132).

6.2.15.4 Example

```
                         HEADING           HOL(OBJECT SOURCE
                                            LOCATION)
```

6.2.16 LISTING

Label field	Command field	Argument field
	LISTING	D,I,C D,I,C...D,I,C

6.2.16.2 Where

Label field -- not used
Command field -- LISTING
General field --

1. Designator

   A. LINE = Line number (in decimal).

   B. LOCATION = Current value of location counter (in hexadecimal or octal).

   C. OBJECT = Object value of GEN, EQU, LIST, DATA or instruction statement (in hexadecimal or octal).

   D. SOURCE = 80 character source line or META expansion line.

   E. SFLAG = Skip flag

   F. EFLAG = Error flag

   G. VALUES = Special user specified value

2. Initial character position GF (2...N,1) = Initial character position of listing field (left justified).

3. Number of characters GF(2...N,2) = Number of characters for this field. Note that all designators' values are right justified, except for source. The number of characters for source cannot exceed 80, and the initial character from the start of the source record is left justified. The user should avoid character position 0, since this is often used for print control.

6.2.16.3 Function

The LISTING directive allows the user to format the DUAL listing to fit particular installation requirements and formats. The LISTING directive is not required, and if not used then the format is as described in Chapter 8. Since the LISTING directive requires general fields, the user may need a LENGTH statement prior to a listing directive. If a LISTING directive is used, then each DUAL listing output line will contain only those fields specified by the LISTING statement. The LISTING directive is very useful in conjunction with the VALUES directive (for machines with extensive object fields,) or where a printer has less than 132 characters (used by the default DUAL printout).

Special Comments: The number of characters listed per record shall contain the rightmost defined column (maximum sum of GF(X,1) and GF(X,2)).

6.2.16.4 Example

Assume that is is desirable to have the DUAL listing output be generated to a teletype with only 72 characters.

```
LISTING       EFLAG,1,1     ;
              LOCATION,8,7 ;
              OBJECT,16,8   ;
              SOURCE,25,47
```

The X's above indicate continuation.

Note this will cause 72 characters to be listed for all DUAL listing lines including title, subtitle and heading lines. The specification of both start column and length allows mutually exclusive fields to occupy the same listing position. The following should also be noted:

- Numbering begins with 0.

- If both VALUES and OBJECT are specified, then only one will be listed. VALUES will be listed within a META at META processing termination.

- Each designator field can only be listed once per line. If specified more than once in a LISTING directive, then the latter specification will prevail.

6.2.17 VALUES

6.2.17.1 Format

Label field	Command field	Argument field
	VALUES	Type,Exp,Exp,Exp

6.2.17.2 Where

Label field -- not used
Command field -- VALUES
Argument field -- The argument field contains four subfields as defined below:

1. AF(0) = Type (C=character, X=hexadecimal, O=octal, D=decimal)

2. AF(1) = Value to generate, if character string must then be enclosed in parentheses.

3. AF(2) = Initial character position.

4. AF(3) = Number of characters (if void then equal to 1).

6.2.17.3 Function

The VALUES directive enables the user to set the VALUES field for listing output. The VALUES directive works in conjunction with the LISTING and HEADING directives.

Numerics are right justified with leading zeroes suppressed. If the numeric field magnitude is greater than the field length, truncation will occur on the left (high order numerics). Textual strings will be left justified. If the textual string is longer than the length, truncation will occur on the right. The VALUES buffer will be cleared (blanks) after print out (META termination).

6.2.17.4 Examples

```
VALUES    C,(GOOD),0,4
VALUES    D,50*30,8,5
VALUES    0,1907-ALPHA,12,3
VALUES    X,HEX(AC),18,2
```

After the above statements, the VALUES field would contain (assuming ALPHA=1901):

```
GOOD_____1500_6___AC
where _ is a blank
```

6.2.17.5 Note

If parentheses are required in a text string, then the same rules which apply to the HOL and C functions apply to the text VALUES value. It should be noted that characters are left justified and all other values are right justified. The user can utilize as many VALUES statements within a META as he desires. Since the VALUES directive is only applicable when a META is processed, overlapping VALUES specifications will contain only the most recent results.

6.2.18 VERSION

6.2.18.1 Format

Label field	Command field	Argument field
	VERSION	Character string

6.2.18.2 Where

Label field -- not used
Command field -- VERSION
Argument field -- Any character string of which only the first 8 characters are used.

6.2.18.3 Function

The VERSION directive is used to specify the Version for the relocatable load module initial record. If more than 8 characters are specified only the first 8 characters are utilized. The linkage editor saves the initial version specification, with subsequent version specifications being ignored. The VERSION directive should appear prior to any other directives that generate object (except the MODULE directive).

6.2.18.4 Examples

```
VERSION        V77
VERSION        IBM370
VERSION        PSS08A
```

6.2.19 MODULE

6.2.19.1 Format

Label field	Command field	Argument field
	MODULE	Symbol

6.2.19.2 Where

Label field -- not used
Command field -- MODULE
Argument field -- A symbol of which only the first 8 characters are used.

6.2.19.3 Function

The MODULE directive has a twofold purpose. The initial MODULE object record encountered by the linkage editor is used for the initial record of the relocatable load module, and the MODULE record is also used to identify the name of a module to be used in conjunction with the INCLUDE and IGNORE commands of the linkage editor. The MODULE directive should appear prior to any other directive that generates object.

6.2.19.4 Examples

```
MODULE          MATHSUBS
MODULE          SORT
MODULE          AZARIA
```

6.2.20 BI#DEC

Label field	Command field	Argument field
	BI#DEC	Expression

6.2.20.2 Where

Label field -- not used
Command field -- BI#DEC
Argument field -- An evaluatable expression which results in a value of 0 thru 3.

6.2.20.3 Function

The BI#DEC directive is used in conjunction with the LISTING directive. It is used to determine whether the location counter in the special listing should be displayed in decimal or hexadecimal (octal if PRINT=0). The default mode is hexadecimal (octal). A value of 1 designates decimal and a value of 0 designates hexadecimal (octal).

6.2.20.4 Examples

```
BI#DEC     1     printout location in decimal
BI#DEC     0     printout location in hexadecimal
```

6.2.21 PRINTVAL

6.2.21.1 Format

Label field	Command field	Argument field
	PRINTVAL	Expression

6.2.21.2 Where

Label field -- not used by this directive
Command field -- PRINTVAL
Argument field -- A evaluatable expression which results in a value of 0 or 1. If void a default value of 0 will be assumed.

6.2.21.3 Function

This directive controls the printing of the VALUES buffer and the META call line which contains this directive. This directive is used in conjunction with the LISTING, HEADER and VALUES directives. It may only be used from within a META.

If printing is enabled (meta not nested or previous meta level print control not 0, 5, or 6) then the current contents of the VALUES buffer will be printed; furthermore, if the argument field expression resolves to a value of one, the call line of this meta will be printed on the same line as the VALUES buffer. It is the user's responsibility to set the value of the location count in the VALUES buffer. It is presumed that the user's LISTING directive allocates a VALUES field that overlaps the OBJECT field.

This directive allows the user to create metas that essentially act as data directives or built-in instructions in which the object generation and listing format is under user control.

The VALUES buffer will be reset to all blank, following printing if enabled, by this directive.

6.2.21.4 Example

```
Values
Field                   Source

       #HEX             META      0
       $HEX1            SET       1
       $HEX2            LOOP      0,1,NUM(2)-1
                        VALUES    X,HEX(AF($HEX2)),0,4
                        PRINTVAL         $HEX1
       $HEX1            SET       0
                        DATA,2    HEX(AF($HEX2))
                        LOOPTEST
                        MEND
1234                    #HEX      1234,5678,9ABC
5678
9ABC
       OUTER            META      OUTER#LIST
                        #HEX      1234,5678
                        #HEX      9ABC
                        MEND
       OUTER#LIST       SET       0
                        OUTER
       OUTER#LIST       SET       2
                        OUTER
1234                    #HEX      9ABC
                        END
```

The above shows an example of a meta that generates 2 bytes for each argument; the latter is assumed to be in hexadecimal. It shows a call from level zero, and calls from within a meta (OUTER) whose list control is first zero and then two.

6.2.22 LIBNAME

6.2.22.1 Format

Label field	Command field	Argument field
	LIBNAME,Expression	symbol

6.2.22.2 Where

Label field -- not used
Command field -- CF(0) must contain LIBNAME. CF(1) must contain an evaluatable expression which results in 0 or 1 or CF(1) must be void (same as 0).
Argument field -- The argument field must contain a legal DUAL symbol.

6.2.22.3 Function

The LIBNAME directive is used to generate object designating the primary (a value of 0 or void in CF(1)) or secondary object library name.

6.2.22.4 Error Conditions

- Illegal symbol
- Illegal expression

6.2.22.5 Examples

```
LIBNAME         MATHLIB
LIBNAME,1       POLYNOM
```

6.2.23 DEBUG

6.2.23.1 Format

Label field	Command field	Argument field
	DEBUG	Exp,Exp,...Exp

6.2.23.2 Where

Label field -- not used
Command field -- DEBUG
Argument field -- The argument must contain one or more subfields, each of which must contain an evaluatable expression.

6.2.23.3 Function

The DEBUG directive provides a mechanism for META's to generate DUAL debug object. This statement is only valid when a special object translator (not the standard DUAL Link Editor) is available.

6.2.23.4 Error conditions

- *Illegal expression*

6.2.23.5 Examples

```
DEBUG          HOLC(AF(I)),5,AF(2)*AF(3)
DEBUG          1,2,3
```

6.2.24 OUTPUT

6.2.24.1 Format

Label field -- not used
Command field -- OUTPUT
Argument field -- contains any characters enclosed in parentheses. The character string must be less than or equal to 80 characters and cannot contain any parentheses. The rules for representing parentheses are identical to the rules for specifying text string constants.

6.2.24.3 Function

The OUTPUT directive enables META writers to output their own text messages from within a META.

6.2.24.4 Error conditions

- Illegal message
- Illegal expression
- Illegal index

6.2.24.5 Examples

```
OUTPUT    (NUMBER OF SUBROUTINES=),SUBCNT,20
OUTPUT    (WARNING-ILLEGAL USAGE OF !),#X,10
```

6.2.25 SPUNCH

6.2.25.1 Format

Label field	Command field	Argument field
	SPUNCH	character,exp,exp

6.2.25.2 Where

Label field -- not used
Command field -- SPUNCH
Argument field --

AF(0) = character to be changed

AF(1) = value to be used for first character (must be between 0 and maximum character value).

AF(2) = value to be used for second character (optional).

6.2.25.3 Function

The SPUNCH directive works in conjunction with the PUNCH-PEND directive to automatically modify a specific character to a different character or two characters for source output.

6.2.25.4 Example

SPUNCH ?,HEX(68)

6.3 LOCATION COUNTER DIRECTIVES

6.3.1 GENERAL

The location counter directives are used to control the location counter, storage allocation and the mode of object (absolute or relocatable) to be generated. Via location counter directives, a programmer can maintain complete control over where both the data and program instructions are to be allocated.

6.3.2 ABSOLUTE

6.3.2.1 Format

Label field	Command field	Argument field
	ABSOLUTE,exp,exp,exp	

6.3.2.2 Where

Label field -- not used
Command field -- ABSOLUTE must appear in CF(0). CF(1) may contain a type code (0-15). CF(2) may contain an initialization indicator (0-3). CF(3) may contain a starting location mode (0-7).
Argument field -- not used

6.3.2.3 Function

The ABSOLUTE directive specifies that subsequent object is to be assembled with absolute storage addresses. That is, the program will execute at exact locations that are displayed on the listing output. An ORIGIN directive should follow the ABSOLUTE statement to designate the absolute address to which the location counter is to be set.

6.3.2.4 Default Case

- DUAL initializes the object mode to relocatable.

6.3.2.5 Example Usage

Label field	Command field	Argument field
	ABSOLUTE	

Designates that subsequent object is to be assembled in absolute mode.

	ORIGIN	0100

Set location counter to 0100 (see ORIGIN directive)

A	DATA	$

Outputs an object value of 0100 to be loaded at memory address 0100. Symbol A is defined as 0100.

6.3.3 RELOCATE

6.3.3.1 Format

Label field	Command field	Argument field
	RELOCATE,Exp,Exp,Exp	

6.3.3.2 Where

Label field -- not used
Command field -- RELOCATE must appear in CF(0). CF(1) may contain a type code (0-15). CF(2) may contain an initialization indicator (0-3). CF(3) may contain a starting location mode (0-7).
Argument field -- not used

6.3.3.3 Function

The RELOCATE directive specifies that subsequent object code is to be assembled with storage addresses that can operate anywhere in memory (i.e., relocatable). That is, locations within the program are relative to the beginning location of the program. Labels that are defined as equal to the current value of the location counter within a relocatable section are considered relocatable.

6.3.3.4 Default Case

- DUAL initializes the object mode to relocatable.

6.3.3.5 Example Usage

Label field	Command field	Argument field
	RELOCATE	

Designates that subsequent object is relocatable.

	ORIGIN	0100

Set location counter to relocatable 0100.

A	DATA	$

Outputs an object value of relocatable 0100 at relocatable location 0100. Defines Label A as equal to relocatable 0100.

6.3.4 ORIGIN

6.3.4.1 Format

Label field	Command field	Argument field
	ORIGIN	Expression

6.3.4.2 Where

Label field -- The label field may contain a symbol. If a symbol does appear in the label field, then it is defined as the value of the current location counter.
Command field -- ORIGIN
Argument field -- The initial argument subfield must contain an evaluatable expression which results in a positive value that is less than the target machine memory size (see KSIZE directive).

6.3.4.3 Function

The ORIGIN directive is used to set the location counter to a designated location. The location counter's section is not changed.

6.3.4.4 Default Case

- The location counter is initialized by DUAL to zero.

6.3.4.5 Error Conditions

- Illegal expression

6.3.4.6 Example Usage

Label field	Command field	Argument field
	ORIGIN	05000

This sets the location counter to 05000.

Label field	Command field	Argument field
	ORIGIN	100000000

This is an illegal ORIGIN since the value is too large.

6.3.5 RESERVE

6.3.5.1 Format

Label field	Command field	Argument field
Symbol (opt.)	RESERVE	EXPRESSION

6.3.5.2 Where

Label field -- The label field may contain a symbol. If a symbol does appear in the label field, then it is defined as the value of the current location counter.

Command field -- RESERVE
Argument field -- The argument field must contain an evaluatable expression which results in a positive value.

6.3.5.3 Function

The RESERVE directive enables the user to reserve an area of memory within his program. The RESERVE directive increments the location counter by the result of the expression in the argument field.

6.3.5.4 Error Conditions

- Illegal symbol in label field.
- Illegal expression in argument field.

6.3.5.5 Example Usage

Label field	Command field	Argument field
ABC	RESERVE	10

Reserves 10 memory words whose initial word can be referenced symbolically as ABC.

	RESERVE	-1

Illegal to reserve negative number of locations.

6.3.5.6 Special Comments

It should be noted that no data values are generated for reserved areas.

6.3.6 RESEND

6.3.6.1 Format

Label field	Command field	Argument field
Symbol (opt.)	RESEND	EXPRESSION

6.3.6.2 Where

Label field -- The label field may contain a symbol. If a symbol does appear in the label field, then it is defined as the value of the current location counter after the reserve.
Command field -- RESEND
Argument field -- The argument field must contain an evaluatable expression which results in a positive value.

6.3.6.3 Function

The RESEND directive enables the user to reserve an area of memory within his program. The RESEND directive increases the location counter by the result of the expression in the argument field. The RESEND directive is identical to the RESERVE directive except that the symbol is defined after the reserve for the RESEND instead of before for the RESERVE directive.

6.3.6.4 Error Conditions

- Illegal symbol in label field
- Illegal expression in argument field

6.3.7 BOUND

6.3.7.1 Format

Label field	Command field	Argument field
	BOUND	Expression

6.3.7.2 Where

Label field -- not used
Command field -- BOUND
Argument field -- The argument field must contain an evaluatable expression which results in a value between 1 and 2048 inclusively.

6.3.7.3 Function

The BOUND directive adjusts the location counter to a multiple of the specified number of units, if necessary. That is, a BOUND 2 will guarantee an even boundary, a BOUND 3 will advance the location counter (if need be) so that it is divisible by 3 and so forth. A BOUND 1 is treated as a special case, with the location counter being adjusted to an odd number of units, if necessary.

6.3.7.4 Error Conditions

- Illegal expression in the argument subfield.

6.3.7.5 Example Usage

Label field	Command field	Argument field
	BOUND	2

This will guarantee an even boundary.

	BOUND	010

This will adjust the location counter to a multiple of 8.

	BOUND	5+4-9

This is illegal since 0 is not between 1 and 2048.

6.3.8 LITORG

6.3.8.1 Format

Label field	Command field	Argument field
	LITORG	

6.3.8.2 Where

Label field -- not used
Command field -- LITORG
Argument field -- not used

6.3.8.3 Function

The LITORG directive is used to control the location of values in the literal pool. All literals that have been processed up to the LITORG statement will be generated starting at the current location counter value.

6.3.8.4 Example Usage

Label field	Command field	Argument field
	LITORG	

6.3.9 BASE

6.3.9.1 Format

Label field	Command field	Argument field
	BASE	Base Register, Location

6.3.9.2 Where

Label field -- not used
Command field -- BASE
Argument field -- The initial argument subfield, AF(0), must contain an evaluatable expression which results in a positive value between 1 and 15. This value specifies a specific base register. The second argument subfield, AF(1), must contain a symbol that is either an external reference or an address.

6.3.9.3 Function

The BASE directive is used so that DUAL can calculate an address as a base register and a displacement. Base addressing implies that addresses are computed as a displacement from a base. The directive states the base address value that DUAL may assume will be in a base register at object time. The BASE statement does not load the base register with a specific value; rather it is the programmer's responsibility to maintain the base registers. If a programmer changes the value in a base register and wishes DUAL to compute addresses as a displacement from this new value, then it is necessary to tell DUAL of this new value by using another BASE statement.

When the user specifies that an address is to be computed as a displacement, DUAL will compare the address specified against the values specified for the base register. As soon as one is found within the minimum positive displacement distance (see SIZE directive), then a displacement is computed relative to that base register's value. If none are within the legal distance, an error will be generated.

6.3.9.4 Error Conditions

- Illegal expression
- Illegal symbol

6.3.9.5 Example Usage

Label field	Command field	Argument field
	REFER	SORT

Specifies that SORT is an external reference.

	BASE	1,SORT

Tells DUAL that base register 1 will have the value of the external reference SORT.

	BASE	4,PGMSTART

Tells DUAL that base register 4 will have the relocatable address PGMSTART.

6.3.10 RELEASE

6.3.10.1 Format

Label field	Command field	Argument field
	RELEASE	Exp,Exp,....Exp

6.3.10.2 Where

Label field -- not used
Command field -- RELEASE
Argument field -- The argument field may contain one or more subfields, each of which contain an expression which results in a value between 1 and 15.

6.3.10.3 Function

The RELEASE directive specifies that a previously available base register may no longer be used as a base register. The expressions in the argument field designate which base registers are to be released. A RELEASE statement is not necessary when the base address in a base register is to be changed via a subsequent BASE directive.

6.3.10.4 Error Conditions

- Illegal expression

6.3.10.5 Example Usage

Label field	Command field	Argument field
	RELEASE	5,6,3,12

Specifies that registers 3,5,6 and 12 are no longer available as base registers.

6.3.11 RESERVUM

6.3.11.1 Format

Label field	Command field	Argument field
Symbol(opt)	RESERVUM	Expression

6.3.11.2 Where

Label field -- The initial label subfield may contain a symbol. If a symbol does appear in the label field, then it is defined as a textual address with a value of the current location counter ($). DUAL will not consider this label as an external definition.

Command field -- The initial command subfield must contain the symbol RESERVUM.

Argument field -- An evaluatable expression that is a positive parameter. Specifies the number of characters to reserve.

6.3.11.3 Function

The RESERVUM directive allows the user to reserve a specified number of characters for a data region. See the DATUM directive for further information.

6.3.11.4 Error Condition

- Illegal symbol
- Illegal expression

6.3.11.5 Example Usage

Label field	Command field	Argument field
ABC	RESERVUM	50

Reserves 50 characters whose first character can be accessed as ABC.

6.3.12 ADJUST

6.3.12.1 Format

Label field	Command field	Argument field
	ADJUST	Expression

6.3.12.2 Where

Label field -- not used
Command field -- ADJUST
Argument field -- An evaluatable expression whose value is between 0 and the number of characters per word minus one.

6.3.12.3 Function

The ADJUST directive allows the user to adjust the location counter to a specified character boundary. See the DATUM directive for further information.

6.3.12.4 Error Condition

- Illegal expression

6.3.12.5 Example Usage

Label field	Command field	Argument field
	ADJUST	0

Adjust location counter to a word boundary.

6.3.13 CSECT

6.3.13.1 Format

Label field	Command field	Argument field
Symbol(opt)	CSECT,exp,exp,exp	

6.3.13.2 Where

Label field -- The initial label field may contain a symbol which names the control section; otherwise the section is considered unnamed (same function as RELOCATE directive).

Command field -- CSECT must appear in CF(0). CF(1) may contain a type code (0-15). CF(2) may contain an initialization indicator (0-3). CF(3) may contain a starting location mode (0-7).

Argument field -- not used

6.3.13.3 Function

The CSECT directive (control section) allows a single program to have multiple control (relocatable) sections. All statements following the CSECT section are assembled as part of that control section until a new section (CSECT or DSECT) statement identifies a new section. A CSECT which has a duplicate name of an already existing control section merely continues the generation of object for that section.

6.3.13.4 Error Conditions

- Illegal symbol
- Maximum number of CSECT/DSECT exceeded (maximum is 32)

6.3.13.5 Example Usage

Label field	Command field	Argument field
STARTPGM	DATA	5,6,7

Generates code for blank CSECT.

ABC	CSECT	

Names new control section ABC.

	DATA	6,7,8

Generates code for section ABC.

	CSECT	
	DATA	HOL(BLANKITY)

Continues code for blank control section.

6.3.14 DSECT

6.3.14.1 Format

Label field	Command field	Argument field
Symbol(opt)	DSECT	

6.3.14.2 Where

Label field -- The initial label subfield may contain a symbol which names the dummy section.
Command field -- DSECT
Argument field -- not used

6.3.14.3 Function

The DSECT directive provides a method for having more than one subroutine referencing the same data. Dummy sections of the same name must be the same size.

6.3.14.4 Error Conditions

- Illegal symbol
- Maximum number of CSECT/DSECT exceeded (maximum is 32)

6.4 SYMBOL DEFINITION AND CONTROL DIRECTIVES

6.4.1 GENERAL

The symbol definition and control directives are used to: assign a symbol to the attributes of an expression or string of expressions, control the sequence of lines to be processed; and declare local symbol regions. Often the programmer will want to assign values to symbols and refer to the value by using the symbol rather than continually using the value throughout the program. This is commonly termed parametric programming. DUAL provides the programmer the necessary tools to perform parametric coding, thereby enabling a programmer to write a program that is easier to understand, modify and maintain.

6.4.2 LABEL

6.4.2.1 Format

Label field	Command field	Argument field
	LABEL	

6.4.2.2 Where

Label field -- not used
Command field -- LABEL
Argument field -- not used

6.4.2.3 Function

The LABEL directive is used to place a label on a statement line for the purpose of process time transfer of control. The LABEL directive is also used in conjunction with the WITHHOLD-WEND directives. This directive operates in all other respects as a NO-OP directive.

6.4.2.4 Example Usage

Label field	Command field	Argument field
	LABEL	

This statement performs no function.

	GOTO	$100

This will cause subsequent lines to be ignored by the DUAL processor until "$100" is found in the label field of a subsequent statement.

	DATA	5
	DATA	6,$

These two lines will be skipped (treated like comments) due to the previous GOTO statement.

$100	LABEL	

DUAL will no longer skip lines since the symbol in the label field of the LABEL directive is the same as the symbol in the argument field of the GOTO directive.

6.4.3 EQU

6.4.3.1 Format

Label field	Command field	Argument field
Symbol	EQU	Expression

6.4.3.2 Where

Label field -- The initial label field must contain a symbol.
Command field -- EQU
Argument field -- The initial argument subfield must contain an expression whose value results in an absolute or relocatable quantity.

6.4.3.3 Function

The EQU directive is used to define or redefine a symbol by assigning it the value of the expression in the argument field. A symbol may be redefined via an EQU directive provided that it was previously a parameter. The EQU directive provides a convenient method for equating a symbolic label to a value of an expression, intermediate data, a counter, a relocatable address or a parameter.

6.4.3.4 Error Conditions

- Illegal or missing symbol in the label field.
- Illegal expression in the argument field.
- Attempt to redefine symbol that is not a parameter.

6.4.3.5 Example Usage

Assume the location counter has a value of 0100 and the relocatable mode is in effect.

Label field	Command field	Argument field
A	EQU	5

Assigns a value of 5 to symbol A (type=parameter).

B	EQU	6*5-8*(7/2-1)

Assigns a value of 14 to symbol B (type=parameter).

	DATA	A,B

Generates two object words, the first with a value of 5 and the second with a value of 14.

X	EQU	$

Assigns the current value of the location counter (0102) to the symbol X. Since the object mode is relocatable, symbol X is given a classification as a relocatable address.

	DATA	X

Generates an object word with a value of 0102 which is relocatable.

A$LONG$TAG	EQU	$-X

Assigns a value of 1 to the symbol "A$LONG$TAG". Note that the type is parameter since the difference of two relocatable addresses gives an absolute value.

6.4.4 LOCAL

6.4.4.1 Format

Label field	Command field	Argument field
	LOCAL	

6.4.4.2 Where

Label field -- not used
Command field -- LOCAL
Argument field -- not used

6.4.4.3 Function

LOCAL symbols are those symbols that are defined and recognized only within a specific area of a program (as opposed to global symbols which are recognized throughout the program). Differentiation between local and global symbols is made on the basis of the symbols' initial character. Any symbol beginning with a $ is considered a local symbol. Local symbols are used for two purposes:

- To avoid the problem of duplicate symbols.
- To enable larger programs to be translated in minimal configurations.

The LOCAL directive clears the entire local symbol stack. The LOCAL directive should not be used within META's, since each META maintains its own local symbol region. It should be noted that there are special considerations when a local symbol is used to calculate an address as a displacement from a base register. If the BASE directive is used, DUAL will ignore all subsequent LOCAL directives and local symbols will be created as if they were global.

6.4.4.4 Example Usage

Label field	Command field	Argument field
$1	EQU	5

Defines local symbol "$1" as equal to 5.

	LOCAL	

Clears local symbol stack.

$2	EQU	$1

This statement is illegal since $1 is no longer defined and therefore the argument is no longer an evaluatable expression.

6.4.5 LIST

6.4.5.1 Format

Label field	Command field	Argument field
Symbol, Expression	LIST	Elem,Elem....Elem

6.4.5.2 Where

Label field -- The initial label subfield must contain a symbol. If only a particular element is to be redefined within a previously defined list, then LF(1...N) should contain a set of subscripts indicating which element.
Command field -- LIST
Argument field -- The argument field must contain one or more elements. An element can be either a parameter, an address, an external, or a NULL specification. In addition, an element can be subdivided into sub-elements by the use of parentheses. All unspecified elements are considered to be null elements. An element can be specified as NULL by the absence of an expression (e.g.$,,6*5...The element between the $ and 6*5 is NULL).

6.4.5.3 Function

The LIST directive enables the user to define a symbol (or redefine an already existing LIST) by assigning it the attributes of a series of values that appear in the argument field. Each expression is considered an element of the LIST. If each element is a parameter and positive and if the total of all elements is less than or equal to the current word size of the target machine (see SIZE directive), then the LIST is considered a machine list and can be used in defining instructions or in generating values within specified bit patterns (see GEN directive). A list is a chain of sequential elements. The Nth element of a list is designated by using the proper index for that element. The initial element of a linear list Z is referenced as Z(0). A particular element of an already existing list may be redefined by including the element indexes beginning in LF(1).

6.4.5.4 Error Conditions

- Illegal symbol
- Symbol already exists as non-list
- Illegal expression in argument subfield
- Illegal list element

6.4.5.5 Example Usage

Label field	Command field	Argument field
L1	LIST	9,5,10

Declares a three element list named L1, where:

L1(0)=9
L1(1)=5
L1(2)=10

STRING	LIST	-7,5,065-HEX(1Z),$+5

Declares a four element list name STRING, where :

STRING(0)=-7
STRING(1)=5
STRING(2)=27
STRING(3)=$+5

STRING,1	LIST	5*STRING(0)

Redefines STRING(1)=-35.

Assume EXT is an external (declared via a REFER statement) and that ADDRESS is an address.

Label field	Command field	Argument field
MAR	LIST	$,(EXT,5,$-5),,(ADDRESS, (EXT,,(5*10,(3,1),2)))

Declares a four element list named MAR, where:

```
MAR(0)=$
MAR(1)=(EXT,5,$-5)
MAR(1,0)=EXT
MAR(1,1)=5
MAR(1,2)=$-5
MAR(2)=NULL
MAR(3)=(ADDRESS,(EXT,,(5*10,(3,1),2)))
MAR(3,0)=ADDRESS
MAR(3,1)=EXT,,(5*10,(3,1),2)
MAR(3,1,0)=EXT
MAR(3,1,1)=NULL
MAR(3,1,2)=(5*10,(3,1),2)
MAR(3,1,2,0)=5*10
MAR(3,1,2,1)=(3,1)
MAR(3,1,2,1,0)=3
MAR(3,1,2,1,1)=1
MAR(3,1,2,2)=2
```

All other elements are null. Assuming the same list, the user can redefine any element he desires, or the entire list. For example:

```
MAR,1          LIST          1
MAR,3,1        LIST          (5,6)
```

would define the MAR LIST to be

```
MAR            LIST          $,1,,(ADDRESS,(5,6))
```

The user of the LIST directive should be aware that DUAL allocates two units for each value plus a control word for parentheses and element counts. As a user redefines a list, if the new list is the same size or less, DUAL will use the existing LISTs space. If it is greater, then DUAL will search for a LIST space that is currently unused and adequate to meet the needs of the newly modified list. If none is available, then DUAL merely allocates additional space for the list (as if it were a new list).

The following example will help illustrate the use of the EL and NL and DEP functions (assuming original MAR list).

```
                              LISTMODE      2

*    Expands all list even outside META's. Note that this
*    will slow DUAL's assembly mode significantly.

          X                   LIST          EL(MAR)

*    Defines X LIST same as MAR list.

          Y                   LIST          0,EL(X),0

*    This defines Y list with 5 elements with initial and
*    final elements of 0 prior to and after MAR list as
*    shown below:

     0,$,(EXT,5,$-5),,(ADDRESS,(EXT,,(5*10,(3,1),2)),0

                              DATA          DEP(X)

*    Results in a value of 1.

                              DATA          DEP(X,0)

*    Results in a value of 1.

                              DATA          DEP(X,1)

*    Results in a value of 2.

                              DATA          DEP(X,1,0)

*    Results in a value of 1.

                              DATA          NL(X)

*    Results in a value of 4.

                              DATA          NL(Y)

*    Results in a value of 6.
```

6.4.6 SET

6.4.6.1 Format

Label field	Command field	Argument field
Symbol	SET	Expression

6.4.6.2 Where

Label field -- The initial label field must contain a symbol.
Command field -- SET
Argument field -- The initial argument subfield must contain an expression whose value results in an absolute or relocatable quantity.

6.4.6.3 Function

The SET directive is used to define or redefine a symbol by assigning it the value of the expression in the argument field. A symbol may be redefined via a SET directive provided that it was previously a parameter or an address. The SET directive provides a convenient method for equating a symbolic label to: a value of an expression, intermediate data, a counter, a relocatable address, or a parameter.

6.4.6.4 Error Conditions

- Illegal or missing symbol in the label field.
- Illegal expression in the argument field.
- Attempt to redefine symbol that is not a parameter or an address.

6.4.6.5 Example Usage

Assume the location counter has a value of 0100 and the relocatable mode is in effect.

Label field	Command field	Argument field
A	SET	5

Assigns a value of 5 to symbol A (type=parameter.)

B	SET	6*5-8(7/2-1)

Assigns a value of 14 to symbol B (type=parameter).

	DATA	A,B

Generates two object words, the first with a value of 5 and the second with a value of 14.

X	SET	$

Assigns the current value of the location counter (0101) to the symbol X. Since the object mode is relocatable, symbol X is given a classification as a relocatable address.

	DATA	X

Generates an object word with a value of 0101 which is relocatable.

A$LONG$TAG	SET	$-X

Assigns a value of 1 to the symbol "A$LONG$TAB". Note that the type is parameter since the difference of two relocatable addresses gives an absolute value.

It should be noted that the only difference between the SET and EQU directive is that the WARNING directive has no effect on the SET directive.

6.4.7 STRING

6.4.7.1 Format

Label field	Command field	Argument field
Symbol	STRING	Character string

6.4.7.2 Where

Label field -- The label field must contain a symbol. The symbol can either be a new symbol or a symbol that is already a name of a string definition.
Command field -- STRING
Argument field -- The argument field can contain any desired character string or be blank.

6.4.7.3 Function

The STRING directive allows the user to assign a character string to a symbol and then later retrieve that string by using the VS function within a META. The STRING/VS directive and function provides for saving and manipulating character strings. The maximum character string is 63 characters.

6.4.7.4 Example Usage

Label field	Command field	Argument field
ALPHABET	STRING	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BLANK	STRING	
GF3	STRING	GF(3)

6.4.8 VALUE

6.4.8.1 Format

Label field	Command field	Argument field
Symbol, Expression	VALUE	Exp,Exp....Exp

6.4.8.2 Where

Label field -- The initial subfield must contain a symbol. If a particular value is to be reset, then LF(1) must contain an evaluatable expression to be used as an index to which user VALUE is to be reset.
Command field -- VALUE
Argument field -- The argument field must contain one or more subfields, each of which contains an evaluatable expression that is of type parameter.

6.4.8.3 Function

The VALUE directive allows a user to build his own symbol table in parallel with DUAL. The user can assign a string of values to a symbol, and later access those values via the UV function. The combination of the VALUE directive and the UV function provides an extremely powerful tool for generating higher order languages. A symbol's user value may be redefined (by using a VALUE directive) as often as need be. The symbol will be processed as command or non-command symbol based on the DUALPROC directive.

6.4.8.4 Default Case

The user value of a new symbol is initialized to zero.

6.4.8.5 Error Conditions

- Illegal symbol in label field.
- Symbol in label field does not exist in symbol table.
- Illegal expression in argument or label field.

6.4.8.6 Example Usage

Label field	Command field	Argument field
ALPHA	EQU	5

Defines symbol "ALPHA" equal to 5.

Label field	Command field	Argument field
ALPHA	VALUE	2,3,6

Defines user value of "ALPHA" as equal to 2,3,6.

BETA	EQU	UV(ALPHA)

Defines symbol "BETA" as equal to 2.

ALPHA,1	VALUE	7

Redefines user values of ALPHA to be 2,7,6.

ALPHA,0	VALUE	UV(ALPHA)+UV(ALPHA,1)

Redefines user values of ALPHA to be 9,7,6.

6.4.8.7 Special Comments

The VALUE directive and user value function were designed to be used within META's so that a META language could classify its own symbols, and then based on this classification, generate the proper object for the META call line.

6.4.9 RENAME

6.4.9.1 Format

Label field	Command field	Argument field
Symbol	RENAME	Symbol

6.4.9.2 Where

Label field -- The label field must contain a new symbol.
Command field -- RENAME
Argument field -- The argument field must contain a previously defined symbol or a DUAL reserved symbol.

6.4.9.3 Function

The RENAME directive allows a user to rename DUAL directives and functions (or his own symbols). The renamed term also continues to be available for usage. Renaming a parameter is the same as using the EQU directive. It should be noted that the symbols associated user values are not copied for a renamed symbol. The symbol will be processed as command or non-command type based on the status of the DUALPROC indicator.

6.4.9.4 Error Conditions

- Illegal symbol
- Argument field symbol has not previously been defined nor is it a DUAL reserved symbol.

6.4.9.5 Example Usage

Label field	Command field	Argument field
SAME$AS	RENAME	EQU

Allows user to equate symbol by using "SAME$AS" command instead of "EQU".

ALPHA	SAME$AS	5*3

Defines symbol ALPHA as equal to 15.

6.4.9.6 Special Comments

The following reserved symbols cannot be renamed: AF,CF,LF,VS,GF,NC, SC,C#, and CH.

6.4.10 ALIAS

6.4.10.1 Format

Label field	Command field	Argument field
Symbol	ALIAS	Symbol

6.4.10.2 Where

Label field -- The label field must contain a new symbol.
Command field - ALIAS
Argument field -- The argument field must contain a previously defined symbol or a DUAL reserved symbol.

6.4.10.3 Function

The ALIAS directive allows a user to change the name of his own symbols, as well as the names of DUAL directives and functions. Unlike RENAME, the previous name becomes undefined and only the new name is available for future usage. The symbol will be processed as command or non-command type based on the status of the DUALPROC indicator.

6.4.10.4 Error Conditions

- Illegal symbol
- Argument field symbol has not previously been defined nor is it a DUAL reserved symbol.

6.4.10.5 Example Usage

Label field	Command field	Argument field
TEST	ALIAS	JUMPSYM

Allows the user to utilize the term TEST instead of JUMPSYM.

	TEST	AF(0),NQ,ALT,$3

This statement causes a transfer to $3 if the symbol in AF(0) is not equal to the symbol ALT.

6.4.10.6 Special Comments

ALIAS causes the symbol in the argument field to become undefined. Consequently, the symbol can be given a new definition.

The following reserved symbols cannot be aliased: AF,CF,LF,VS,NC,SC, C#, and CH.

6.4.11 DUALPROC

6.4.11.1 Format

Label field	Command field	Argument field
	DUALPROC	Expression

6.4.11.2 Where

Label field -- Not used
Command field -- DUALPROC
Argument field -- The initial argument subfield must contain an evaluatable expression which results in a value of either 0 or 1.

6.4.11.3 Function

The DUALPROC directive enables the user to control the processing of the RENAME, ALIAS and VALUE directives, and the UV and TYPE functions. When AF(0)=0 the aforementioned directives and functions will be subsequently processed such that the symbols are handled as non-command type. AF(0)=1 designates that symbols are to be treated as command type.

6.4.11.4 Default Case

DUAL will process RENAME, ALIAS, VALUE, UV and TYPE as non-command type symbols.

6.4.11.5 Error Conditions

- Illegal expression in argument subfield.
- Argument subfield value not equal to 0 or 1.

6.4.11.6 Example Usage

Label field	Command field	Argument field
ALPHA	EQU	5
	DUALPROC	0
ALPHA	VALUE	1

At this point ALPHA has been assigned a user value of 1 since the DUALPROC directive specified non-command type symbol processing for the VALUE directive. Had non-command type symbol processing not been in effect, the VALUE directive would have been found in error due to ALPHA existing only as a non-command type symbol.

	DUALPROC	1
SAME$AS	RENAME	EQU

Since the DUALPROC directive specified command type symbol processing for the RENAME directive, SAME$AS is now defined as a command type symbol whose function is equivalent to the EQU directive. Had command type symbol processing not been in effect the RENAME directive would have been found in error since EQU is a command type symbol.

6.4.12 STATUS

6.4.12.1 Format

Label field	Command field	Argument field
SYMBOL	STATUS	SYMBOL[,VALUE]...

Where,

LF(0)=A legal DUAL symbol.
CF(0)=STATUS
GF(I,0)=SYMBOL
GF(I,1)=Optional value.
(where I=2...n)

6.4.12.3 Function

The STATUS directive is used to assign a set of status symbols to a set of values. At the start of the directive, the current status value is initialized to zero. This value is incremented by one for each subsequent field, unless a value is specified for a field. If a value is specified, then that value is now used as the current status value. The STATUS directive concatenates the label field symbol, separated by a $, with the initial symbol in general field 2 through N. This newly formed symbol is assigned the current status value.

6.4.12.4 Example

Label field	Command field	General fields
SEX	STATUS	MALE FEMALE

this would create two new symbols,

SEX$MALE=0
SEX$FEMALE=1

AIRPLANE	STATUS	B747,3 L1011,5 DC10

this would create three new symbols,

AIRPLANE$B747=3
AIRPLANE$L1011=5
AIRPLANE$DC10=6

6.5 DATA GENERATION DIRECTIVES

6.5.1 GENERAL

The data generation directives are used to represent data conveniently within a symbolic program. Unlike conventional assemblers, the user does not have to specify via the directive what type of constant he is generating. Use of a symbol is optional in the label field. If a symbol does appear in the label field, then it is defined as the value of the current location counter. If the statement appears within a relocatable section, then the symbol is considered a relocatable address.

6.5.2 DATA

6.5.2.1 Format

Label field	Command field	Argument field
Symbol (opt.)	DATA,Exp(opt.)	Exp,Exp....Exp

6.5.2.2 Where

Label field -- The initial label subfield may contain a symbol. If a symbol does appear in the label field, then it is defined as the value of the current location counter.
Command field -- The initial command subfield, CF(0), must contain the symbol "DATA". The second subfield within the command field, CF(1), may contain an expression to indicate the multiple number of addressable units, when different than the word size, of object code to be generated. The maximum is a full word of object. For example, on the IBM 360 with a word size of 32 bits and an address size of 8 bits, CF(1) may contain a 1, 2, 3, or 4 indicating tht 8, 16, 24, or 32 bits should be generated, respectively. A blank CF(1) causes a full word to be generated.
Argument field -- The argument field must contain one or more subfields, each of which contains a legal expression.

6.5.2.3 Function

The DATA directive enables the programmer to represent data conveniently within a program. The data may take the form of an arithmetic expression, a floating point constant, a fixed point constant, a textual constant, a decimal constant, an octal constant, a binary constant, a hexadecimal constant, a relocatable address, an external reference, a forward reference or certain combinations of the aforementioned.

6.5.2.4 Error Conditions

- Illegal label field
- Illegal expression within a subfield

6.5.2.5 Example Usage

Assume we are in a relocatable section with the current value of the location counter equal to 0100.

Label field	Command field	Argument field
A	DATA	5

This would define symbol "A" as a relocatable address at location octal 100. It would also generate 16 bits of object that contains a value of 5.

B	DATA	HOL(ABCDEFGHIJKLMNOPX QRSTYVWXTZ 0123456789/L/R)

This would generate a series of object words beginning at location 0101. The number of words generated would depend on the number of characters per word in the target machine. Symbol B would be defined as a relocatable address equal to 0101.

Label field	Command field	Argument field
	DATA	$

Assuming we are now at location 0110, this statement would generate an object word with a relocatable value of 0110.

	DATA	5*6,A,B,B-A

This statement would generate four object words. The first would contain a value of 30, the second would contain a relocatable value of 0110, the third would contain a relocatable value of 0101 and the fourth would generate a value of one (note that B-A is not relocatable).

LF(0)	DATA	AF(I)

Assume this line is within a META definition and that "I" is equal to a value of 1. Further, assume that this statement appears in a META defined as "ABC". If the META call line were ALPHA ABC 5,6,7 then the example (above) would be processed as if it were: ALPHA DATA 6.

123	DATA	5

This input statement would be flagged as an error since the label field contains an illegal symbol.

6.5.3 GEN

6.5.3.1 Format

Label field	Command field	Argument field
Symbol (opt.)	GEN, machine list symbol	Exp,Exp,....Exp

6.5.3.2 Where

Label field -- The initial label subfield may contain a symbol. If a symbol does appear in the label field, then it is defined as equal to the current location counter.
Command field -- The initial command field, CF(0), must contain the symbol "GEN". The second subfield within the command field, CF(1), must contain a symbol which is the label of a machine list directive.
Argument field -- The argument field may contain one or more subfields, each of which contains a legal expression. If the GEN statement is within a META definition, then a field function (AF,CF,LF,GF,GFA,AFA,CFA) is legal.

6.5.3.3 Function

The purpose of the GEN directive is to generate a specified value list into a designated bit pattern. The user specifies the bit pattern via a machine list symbol and the value list via the argument field. There is a one-to-one correspondence between the machine list and the value list (argument subfields); namely, object is generated with the first list field containing the value of the first argument subfield, AF(0), and so-forth.

6.5.3.4 Error Conditions

- Illegal label field
- CF(1) does not contain a machine list symbol.
- Illegal expression within argument subfield.

6.5.3.5 Example Usage

Assume that the word size of the target machine is 20.

Label field	Command field	Argument field
L1	LIST	2,4,6,8

This statement defines a machine list whose bit pattern looks like

```
0..1...2..5...6..11....12...19
```

ABC	GEN,L1	1,2,3,4

This statement would generate object with value of octal 01101404. Schematically, the object word when placed in the proper bit pattern would look like

```
0...1   2...5   6...11   12...19
  01      02      03       04
```

or in binary format:

```
0...1   2...5   6...11   12....19
  01     0010   000011   00000100
```

Symbol ABC would be defined as equal to the value of the current location counter.

6.5.4 DATUM

6.5.4.1 Format

Label field	Command field	Argument field
Symbol(opt.)	DATUM,exp(opt.)	Expression

6.5.4.2 Where

Label field -- The initial label subfield may contain a symbol. If a symbol does appear in the label field then it is defined as a textual address with a value of the current location counter. DUAL will not consider this label as an external definition.
Command field -- The initial command subfield must contain the symbol DATUM. CF(1) may contain an evaluatable expression which results in a value between 0 and the number of characters per word in the target computer.
Argument field -- The initial argument field must contain a legal expression. A negative expression will cause truncation from the most significant portion of the word.

6.5.4.3 Function

The DATUM directive is used to generate textual constants.

6.5.4.4 Error Conditions

- Illegal label field
- Illegal expression

6.5.4.5 Example Usage

Label field	Command field	Argument field
TEXTADRI	DATUM,3	HOL(ACDEFG)

Generates a textual constant.

6.6 TARGET MACHINE CHARACTERISTIC DIRECTIVES

6.6.1 GENERAL

The target machine characteristic directives are used to describe the target machine to the DUAL processor. If the characteristics of the target machine are the same as that which the DUAL processor is operating on, then there is no need to use the target machine characteristic directives.

6.6.2 KSIZE

6.6.2.1 Format

Label field	Command field	Argument field
	KSIZE	Expression

6.6.2.2 Where

Label field -- not used
Command field -- KSIZE
Argument field -- The initial argument subfield must contain an evaluatable expression which results in a positive value.

6.6.2.3 Function

The KSIZE directive enables the user to be able to specify the number of locations for which DUAL will generate object.

6.6.2.4 Default Case

The target machine memory size is initialized to 131,072 addressing units.

6.6.2.5 Error Conditions

- Illegal expression

6.6.2.6 Example Usage

Label field	Command field	Argument field
	KSIZE	1000-9999

Illegal since KSIZE is negative.

6.6.3 SIZE

6.6.3.1 Format

Label field	Command field	Argument field
	SIZE	Exp1,Exp2, Exp3,Exp4

6.6.3.2 Where

Label field -- not used
Command field -- SIZE
Argument field -- The initial argument subfield specifies the word size of the target machine. It must contain an evaluatable expression which results in a value between 8 and the word size of the host machine. The next argument is the address size. If the address size is not specified, then it will be set equal to the word size. The third argument subfield is used to define the page size as a multiple of the address size. This value is used when addresses are to be computed as displacements (see BR and DISP function) and defines the maximum displacement. The fourth argument defines the default data word size. If not specified it is set equal to the target word size.

6.6.3.3 Function

The SIZE directive is used to define the word size, address size, page size, and default data size of the target machine. If a SIZE directive is used, it must appear prior to the generation of any statements which generate object code. Only one size specification may occur per program.

6.6.3.4 Default

The target machine word size, address size and page size are usually initialized to the characteristics of the machine on which the DUAL processor is executing.

6.6.3.5 Error Conditions

- Illegal expression

6.6.3.6 Example Usage

Label field	Command field	Argument field
	SIZE	32,8,4096

Defines a target machine word size of 32, address size of 8 bits and a page size of 4096 (this describes the IBM 360).

	SIZE	16

Defines a target machine word size of 16.

	SIZE	200

Illegal since 200 is not between 8 and the host machine's word size.

6.6.4 FLOAT

6.6.4.1 Format

Label field	Command field	Argument field	General fields
	FLOAT	p,s(e),s(m), n,b,c,f S,IB,NB	S,IB,NB.....

6.6.4.2 Where

Label field -- not used
Command field -- FLOAT
Argument field -- The first seven subfields must contain evaluatable expressions that result in positive values:

p:AF(O) is used to specify the precision of this definition (where 1=single precision, 2=double, 3=triple, 4=quadruple). This precision corresponds to the precision specified in the FL function.

s(e):AF(1) is used to specify the size of the exponent in bits (including sign bit, if any).

s(m):AF(2) is used to specify the size of the mantissa in bits (including sign, if any).

n:AF(3) is used to specify the format for a floating point negative number (0=sign magnitude, 1=1's complement, 2=2's complement, 3=1's complement of mantissa only, 4=2's complement of mantissa only, 5=sign magnitude mantissa, 2's complement exponent).

b:AF(4) is used to specify the characteristic base displacement (Exp. field -B=power for exponent).

c:AF(5) is used to specify exponent characteristic (0=2, 1=8, 2=16).

f:AF(6) is used to specify the number of general fields to follow.

General fields -- Each of the general fields is made up of three subfields. (Note: See LENGTH directive)

GF(N,0) is used to specify the source of data to be transferred into the floating point data structure (M=mantissa, E=exponent, S=sign, C=constant).

GF(N,1) is used to specify where to get the first bit to be transferred (for "M" or "E" in GF(N,0), GF(N,1) must be a positive evaluatable expression which indicates the initial bit position of the given field to be transferred, for C in GF(N,0), GF(N,1) is the constant value to be transferred).

GF(N,2) is used to specify how many bits are to be transferred sequentially starting where GF (N,1) specified and transferring GF(G,2) bits--GF(N,2) must be a positive evaluatable expression.

6.6.4.3 Function

The FLOAT directive enables the user to specify the target machine's format for floating point numbers in single, double, triple and quadruple precision if necessary.

6.6.4.4 Default Case

The floating-point format of the host computer on which DUAL is operating.

6.6.4.5 Error Conditions

- Illegal expression
- Missing field or subfield
- Improper description
- Definition does not end in a word boundary
- Staging field overrun

6.6.4.6 Example Usage

Method: The mantissa, from an FL function, is assembled in the staging field-M, left justified and normalized in four words. The high order bit is bit 0, the low order bit is equal to 4 x word size of the host machine - 1. Bit 0 is a sign bit when one is called for. The binary point is assumed between 0 and 1.

The exponent from an FL function is assembled in the staging field-E, right justified in one word. The high order bit is bit 0, the low order bit is equal to the word size -1. The sign bit is bit 0 when one is called for.

Each constant is assembled in a one word staging field right justified. The high order bit of the transferred constant is in word size - 12 - AF(2). The low order bit of the transferred constant is in the bit word size - 1.

The floating point data structure is filled in with transferred bits serially from high order to low order bit positions.

Single Precision Floating point word format

number = mantissa x 2(exp) - 64
negnumber = 2's complement of positive representation

Label field	Command field	Argument field	General fields	
	FLOAT	1,8,16,2,	M,0,16	E,16,1
		64,2,4	C,0,8	E,17,17

6.6.5 FIXED

6.6.5.1 Format

Label field	Command field	Argument field	General field
	FIXED	p,s,n,f	T,F,N...T,F,N

6.6.5.2 Where

Label field -- not used
Command field -- FIXED
Argument field -- The first four subfields must contain an evaluatable expressions which result in positive value:

p:AF(0) is used to specify the precison of this definition (where 1=single precision, 2=double, 3=triple, 4=quadruple). This precision corresponds to the precision specified in the FX function.

s:AF(1) is used to specify the size of the data structure in bits. This must be a multiple of the target machine's word size (4 X maximum).

n:AF(2) is used to specify the format for a negative number. (0=sign magnitude, 1-1's complement, 2=2's complement).

f:AF(3) is used to specify the number of general fields to follow.

General field - Each of the general fields is made up of three subfields. (Note: See LENGTH directive)

GF(n,0) is used to specify the source of data to be transferred into the fixed point data structure (N=number, C=constant).

GF(n,1) is used to specify where to get the first bit to be transfereed (for an "N" in GF(n,0), GF(n,1) must be a positive evaluatable expression which indicates the initial bit position of the given field to be transferred, for "C" in GF(n,0), GF(n,1) is the constant value to be transferred.

GF(n,2) is used to specify how many bits are to be transferred sequentially, starting where GF(n,1) is specified, and transferring GF(n,2) bits. GF(n,2) must be a positive evaluatable expression.

6.6.5.3 Function

The FIXED directive enables the user to specify the target machine's format for fixed point numbers in single, double, triple, or quadruple precision if necessary.

6.6.5.4 Error Conditions

- Illegal expression
- Missing field or subfield
- Improper description
- Definition does not end on a word boundary
- Staging field overrun

6.6.5.5 Example Usage

Method - A fixed point number described by a FX function is assembled in the staging field -N right justified. The staging field is four words long. Bit 0 is a sign bit if one is called for.

Each constant is assembled in a staging field one word long. Constants are right justified. The high order bit of the transferred constant is defined as (word size -1) - AF(2). The low order bit of the transferred bit is word size - 1.

6.6.6 NEGATIVE

6.6.6.1 Format

Label field	Command field	Argument field
	NEGATIVE	Expression

6.6.6.2 Where

Label field -- not used
Command field -- NEGATIVE
Argument field -- The initial argument field must contain an evaluatable expression which results in a value between 0 and 2. The values between 0 and 2 have the following meaning:

0 = sign magnitude
1 = 1's complement
2 = 2's complement

6.6.6.3 Function

The NEGATIVE directive allows the user to specify the format for the target machine's negative numbers.

6.6.6.4 Default Case

The same as the computer on which DUAL is operating.

6.6.6.5 Error Conditions

- Illegal expression

6.6.6.6 Example Usage

Label field	Command field	Argument field
	NEGATIVE	0

Negative numbers output as sign magnitude.

	NEGATIVE	2

Negative numbers output in 2's complement.

6.6.7 PROHOL

6.6.7.1 Format

Label field	Command field	Argument field
LISTNAME	PROHOL	E1,E2,E3,E4

6.6.7.2 Where

Label field -- Name of a list.
Command field -- PROHOL
Argument field -- The argument field must contain four subfields each of which contains an evaluatable expression. The contents of the subfields are:

AF(0) = Value for a word containing all blanks
AF(1) = Number of characters per word
AF(2) = Number bits per character
AF(3) = Rightmost character offset

6.6.7.3 Function

The PROHOL directive enables the user to specify the target machine's character set in terms of the host computer's character set. Using the PROHOL directive, the user may translate his characters from the host machine's character code. If the PROHOL directive is used, then each character (from a HOL, a HOLC, or C function) is used as an index into the list of values specified by the name of the list in the PROHOL statement to obtain a new value.

6.6.7.4 Default Case

DUAL will initially generate characters in the format of the host machine.

6.6.7.5 Error Conditions

- Illegal list in label field
- Illegal expression in argument subfield
- The number of bits per character times the number of characters per word plus the character offset exceeds the target machine's word size.

6.6.7.6 Example Usage

Assume that DUAL is operating on a computer with a 6 bit BCD character set (i.e., "0" =0, "A" = 021, "BLANK" = 060, etc.) and it is desired to change the characters to EBCDIC (i.e., "0"=HEX (F0), "A"=HEX(C1), "BLANK"=HEX(40), etc.). Further assume a LIST named EBCDIC has been declared that has 64 elements, each being a value from 0 to HEX(FF). Then the following PROHOL statement would be used to cause DUAL to automatically translate characters from 6 bit BCD to 8 bit EBCDIC in a 32 bit machine.

Label field	Command field	Argument field
EBCDIC	PROHOL	HEX(40404040),4,8,0

The initial argument AF(0), specifies the contents of a word containing all blanks. The second argument, AF(1), says there are 4 characters per word and the third argument says each character is 8 bits. The zero says characters are not offset.

	DATA	HOL(ABC)

This statement would generate a value of HEX(C1C2C340).

Note that the PROHOL statement causes all subsequent SV function usages to be invalid.

6.7 TARGET MACHINE INSTRUCTION DEFINITION DIRECTIVES

6.7.1 GENERAL

DUAL provides a set of directives which enables a user to easily define any specific set of machine instructions. Since computers have a limited number of instruction formats, the directives are designed such that the user first defines the instruction formats and then specifies the set of instructions which belong to a particular format. Instruction format definitions consist of a LIST that defines the number of bits comprising each field along with a parallel list consisting of functional references for determining what values are to be used in generating the object. Using the target machine instruction directives the user can, in effect, define an assembler for his target machine and still have the ability to utilize the remaining DUAL directives. If addresses are to be computed as displacements (see BR and DISP functions), or if general fields are to be used, then the CMND directive or a meta must be used to define instructions.

6.7.2 LIST (MACHINE LIST)

6.7.2.1 Format

Label field	Command field	Argument field
Symbol	LIST	Exp(1),Exp(2),...,Exp(N)

6.7.2.2 Where

Label field -- The label field must contain a symbol.
Command field -- LIST (machine)
Argument field -- The argument field must contain one or more subfields, each of which contains an evaluatable expression (parameter) which results in a positive value. The sum of the values of each subfield must be equal to or less than the word size of the target machine (see SIZE directive).

6.7.2.3 Function

The machine LIST is a special subset of the normal LIST directive. It enables the user to specify a bit pattern to be used in defining machine instructions (hence a machine list).

6.7.2.4 Error Conditions

- Label field does not contain a legal symbol.
- Symbol already exists.
- Illegal expression in an argument subfield.
- If the sum of the values of the argument subfields is not less than or equal to the word size of the target machine, then this LIST cannot be used in define machine instructions.

6.7.2.5 Example Usage

Assume a target machine with a word size of 32 bits.

```
Label          Command        Argument
field           field          field

SIGMA5         LIST           1,7,4,3,17
```

Defines a machine LIST named SIGMA5 which defines a bit pattern as follows:

```
0  1..7  8..11  12..14  15..31
```

```
Label          Command        Argument
field           field          field

L1             LIST           12,12,8
```

Defines a machine list named L1 which defines a bit pattern as follows:

```
0..11  12...23  24......31
```

```
123            LIST           18,14
```

Illegal statement since label field does not contain a legal symbol.

```
FORMAT$0       LIST           5,$,7
FORMAT$1       LIST           12,6,10,8
```

Defines legal LISTs but not legal machine lists.

6.7.3 STRUCTURE

6.7.3.1 Format

Label field	Command field	Argument field
Symbol	STRUCTURE,List	FR(1),FR(2),...FR(N)

6.7.3.2 Where

Label field - The label field must contain a symbol.
Command field - The initial command subfield, CF(0), must contain the symbol "STRUCTURE". The second subfield within the command field, CF(1), must contain a symbol which is the name of a "machine list".
Argument field -- The argument field must contain one or more subfields, each of which must contain a field function (AF,CF,LF,AFA,CFA) or an "OP" function. The "OP" function is used to specify which field is to contain the operation code.

6.7.3.3 Function

The STRUCTURE directive is used to define an instruction format and to specify the coding format for using that STRUCTURE. Unlike most meta or macro assemblers with target instructions facilities, DUAL provides two distinct advantages via the LIST-STRUCTURE-INST directives:

- The user can specify which subfields are to be used in writing target machine instructions.

- The user no longer has to specify a bit pattern with each instruction definition.

Note that there is a one-to-one correspondence between the elements in the MACHINE LIST and each subfield in the argument field. The initial argument field is used to determine the value of the first field specification of the MACHINE LIST, and so forth.

6.7.3.4 Error Conditions

- Illegal symbol in the label field.
- CF(1) does not contain a MACHINE LIST.
- A subfield within the argument field is not a functional reference.

6.7.3.5 Example Usage

Label field	Command field	Argument field
FMT1	LIST	1,7,4,3,17

Defines a MACHINE LIST whose name is "FMT1".

```
STRUCT1       STRUCTURE,FMT1    AFA(0),OP,CF(1),;
                                AF(1),AF(0)
```

The above STRUCTURE and LIST directive statements specify a format that would be pictured as:

AFA(0)	OP	CF(1)	AF(1)	AF(0)
0	1......7	8.....11	12...14	15.......31

such that an object word would be generated by substituting the appropriate values from the instruction statements that use this format.

6.7.4 INST

6.7.4.1 Format

Label field	Command field	Argument field
Symbol	INST	Expression,Structure

6.7.4.2 Where

Label field -- The initial label subfield must contain a symbol. This is the name of the instruction.
Command field -- INST
Argument field -- The initial argument subfield, AF(0), must contain an evaluatable expression which results in a positive value to be used as the opcode. The second subfield in the argument field, AF(1), must contain a symbol which is the name of a STRUCTURE.

6.7.4.3 Function

The INST directive is used to name instructions of the target machine. The INST directive is used in conjunction with the LIST-STRUCTURE directives in defining a target machine's instruction set.

6.7.4.4 Error Conditions

- Illegal symbol
- Illegal expression
- Argument field one, AF(1), is not the label of a STRUCTURE statement.

6.7.4.5 Example Usage

Assume target machine word size is 32.

```
        Label          Command         Argument
        field           field           field

        ABC            ORIGIN          0100
```

Set location counter = 0100

```
        L1             LIST            1,7,4,3,17
```

Defines machine list named L1.

```
        STRUCT1        STRUCTURE,L1    AFA(0),OP,CF(1),;
                                       AF(1),AF(0)
```

Defines a structure named STRUCT1. See STRUCTURE examples for diagram.

```
        CALL           INST            025,STRUCT1
```

Defines an instruction named CALL with an OP CODE value of 025 and a structure defined by structure STRUCT1.

```
        Label          Command         Argument
        field           field           field

                       CALL            *ABC,5
```

This would generate an object word with a value of 02250240O100. Schematically, the object word when placed in the proper bit pattern would look like:

```
        1         025        0        5       0100
        0       1......7  8...11  12...14  15...31
```

6.7.5 CMND

6.7.5.1 Format

Label field	Command field	Argument field
Symbol	CMND,List	Arg(1),Arg(2),...Arg(N)

6.7.5.2 Where

Label field -- The label field must contain a symbol. This defines the name of the command.
Command field -- The initial command field must contain the symbol "CMND." The second subfield within the command field, CF(1), must contain a symbol which is the name of a machine list.
Argument field -- The argument field must contain one or more subfields, each of which contains a legal expression or a field function (LF,CF,AF,AFA,CFA,GF,GFA,BR and DISP.)

6.7.5.3 Function

The purpose of the CMND directive is to convert a specified value list into a designated bit pattern by simply referencing the command name. In effect, a "CMND" directive enables a user to define instructions with more than one preset value (as opposed to the INST-STRUCTURE concept which allows only for one value--operation code). In addition, a "CMND" directive allows the user to specify a relocatable address as one of the subfields within the argument field. Any label which appears on a command reference will be defined as equal to the current value of the location counter.

6.7.5.4 Error Conditions

- Illegal label field
- CF(1) does not contain a machine list symbol
- Illegal expression within argument subfield

6.7.5.5 Example Usage

Label field	Command field	Argument field
L1	LIST	2,4,6,8

Defines a machine list named L1.

CLA	CMND,L1	0,AF(1),AF(0),7

Defines a "CMND" named CLA.

ALPHA	CLA	5,6

Defines ALPHA as equal to the current location counter and generates a value of 0302407. Schematically, the object word would look like:

```
   0      6      5      7
0...1 2...5 6...11 12...19
```

6.8 PROGRAM MODULE COMMUNICATION DIRECTIVES

6.8.1 GENERAL

A powerful feature of DUAL is the provision for separate processing of inter-dependent programs. The programmer is no longer burdened by the problem of memory allocation. Instead, he is able to refer to distinct modules by name even though they are not processed within his source program. The symbolic inter-program communication that this necessitates is made possible by external symbols. An external reference enables a programmer to refer to symbols in another module, where an external definition enables other modules to refer to a symbol within his source program.

6.8.2 DEFINE

6.8.2.1 Format

Label field	Command field	Argument field
	DEFINE	Symbol(1),Symbol(2) ...Symbol(N)

6.8.2 Where

Label field -- not used
Command field -- DEFINE
Argument field -- The argument field must contain one or more subfields, each of which must contain a legal symbol.

6.8.2.3 Function

The DEFINE directive is used to declare a symbol that is defined in this source program and referenced in other source programs. Symbols declared with a DEFINE directive are used for symbolic program linkage between two or more programs. These symbols are usually the starting point of a subroutine, a data item, or a parameter. External definitions must be DEFINE'd prior to their usage.

6.8.2.4 Error Conditions

- Illegal symbol
- Symbol already declared

6.8.2.5 Example Usage

Label field	Command field	Argument field
	DEFINE	SORT

Identifies the symbol SORT as a symbol that may be referenced in other source programs.

	DEFINE	COTANGENT,SQUARE$ROOT,SIN

Identifies the symbols COTANGENT,SQUARE$ROOT,SIN as symbols that are defined in other source programs.

	DEFINE	5+8

Illegal since argument field is not a symbol.

6.8.3 SUBROUTINE

6.8.3.1 Format

Label field	Command field	Argument field
	SUBROUTINE	Symbol (1), Symbol (2)......Symbol (n)

6.8.3.2 Where

Label field - not used
Command field -- SUBROUTINE
Argument field -- The argument field must contain one or more subfields, each of which must contain a legal symbol.

6.8.3.4 Function

The SUBROUTINE directive is identical to the DEFINE directive except that it classifies the external definition symbol as an entry point to a subroutine.

6.8.3.5 Error Conditions

- Illegal symbol
- Symbol already declared

6.8.3.6 Example Usage

Label field	Command field	Argument field
	SUBROUTINE	SEARCH

Identifies the external definition symbol SEARCH as a subroutine entry point defined within this module.

6.8.4 REFER

6.8.4.1 Format

Label field	Command field	Argument field
	REFER[,S]	Symbol(1),Symbol(2), ...Symbol(N)

6.8.4.2 Where

Label field -- not used
Command field -- REFER must appear in CF(0). An optional S may appear in CF(1) designating a secondary reference.
Argument field -- The argument field must contain one or more subfields, each of which must contain a legal symbol.

6.8.4.3 Function

The REFER directive is used to declare those symbols that are defined in other source programs, but are referenced in this source program. At link time, all symbols that have appeared in a REFER statement must be satisfied by correspoinding external definitions (see DEFINE directive). REFERed variables must be declared prior to their usage.

6.8.4.4 Error Conditions

- Illegal symbol

6.8.4.5 Example Usage

Label field	Command field	Argument field
	REFER	SORT

Declared symbol SORT as an external reference.

	REFER	FLTFIX,MEMA,MATMUL

Declares symbols FLTFIX,MEMA,MATMUL as external references.

	REFER	100

Illegal since 100 is not a symbol.

6.8.5 SEGREF

6.8.5.1 Format

Label field	Command field	Argument field
	SEGREF[,S]	Symbol (1), Symbol (2)......,Symbol (n)

6.8.5.2 Where

Label field -- not used
Command field -- SEGREF must appear in CF(0). An optional S may appear in CF(1) designating a secondary SEGREF.
Argument field -- The argument field must contain one or more subfields, each of which must contain a legal symbol.

6.8.5.3 Function

The SEGREF directive is identical to the REFER except that it classifies the external reference as a segment reference.

6.8.5.4 Error Conditions

- Illegal symbol
- Symbol has already been used

6.8.5.5 Example Usage

Label field	Command field	Argument field
	SEGREF	MATRIX

6.8.6 REFROUTINE

6.8.6.1 Format

Label field	Command field	Argument field
	REFROUTINE[,S]	Symbol (1), Symbol (2)

6.8.6.2 Where

Label field -- not used
Command field -- REFROUTINE must appear in CF(0). An option S may appear in CF(1) designating a secondary REFROUTINE.
Argument field -- The argument field must contain a legal symbol.

6.8.6.3 Function

The REFROUTINE is identical to the REFER directive except that it classifies the symbol as an entry routine.

6.8.6.4 Error Condition

- Illegal symbol
- Symbol already declared

6.8.6.5 Example Usage

Label field	Command field	Argument field
	REFROUTINE	DISTANCE,SINE,COSINE

6.8.7 COMMON

6.8.7.1 Format

Label field	Command field	Argument field
Symbol (opt.)	COMMON,EXP,EXP,EXP	

6.8.7.2 Where

Label field -- The initial label subfield may contain a symbol which names the COMMON section, otherwise the section is treated as "blank" COMMON.
Command field -- COMMON must appear in CF(0). CF(1) may contain a type code (0-15). CF(2) may contain an initialization (0-3). CF(3) may contain a starting location mode (0-7).
Argument field -- not used.

6.8.7.3 Function

The COMMON directive enables a programmer to use an area of memory that is shared by other programs. The COMMON directive is used for reserving space in both "blank" and "labeled" COMMON.

6.8.7.4 Error Conditions

- Illegal symbol

6.8.7.5 Example Usage

Label field	Command field	Argument field
COMMEX	COMMON	

Declares a COMMON section named COMMEX.

6.8.8 CBASE

6.8.8.1 Format

Label field	*Command field*	*Argument field*
	CBASE	Expression

6.8.8.2 Where

Label field -- *not used*
Command field -- CBASE
Argument field -- *an evaluatable expression*

6.8.8.3 Function

The CBASE directive is used to define the starting address for COMMON. The initial "common section" will be assigned this address. A CBASE value of hexadecimal 7FFFFF causes DUAL's linkage editor to ignore the starting COMMON address (hence providing a user-oriented overlay mechanism) and starts common sections merely in the next sequential location.

6.8.8.4 Default Case

The default CBASE value is octal 010000 (hexadecimal 1000).

6.8.8.5 Error Conditions

- Illegal expression

6.8.8.6 Example Usage

Label field	Command field	Argument field
	CBASE	0100
	CBASE	ALPHA
	CBASE	HEX(20000)

6.9 INPUT STATEMENT PROCESSING CONTROL DIRECTIVES

6.9.1 GENERAL

DUAL permits the user to conditionally determine the sequence in which input statements are processed. The input statement processing control directives allow selective generation and skipping of input statements, with character strings, or expressions used to determine which statements are to be processed.

6.9.2 END

6.9.2.1 Format

Label field	Command field	Argument field
	END	Expression (optional)

6.9.2.2 Where

Label field -- not used
Command field -- END
Argument field -- The argument field can contain an evaluatable expression.

6.9.2.3 Function

The END directive terminates the processing of a source program. The END directive can specify a location to which control is to be transferred after the program has been loaded. The END statement must always be the last statement in the source program.

6.9.2.4 Error Conditions

- Illegal expression

6.9.2.5 Example Usage

Label field	Command field	Argument field
	END	

Signifies end of source program.

Label field	Command field	Argument field
	END	ALPHA

Signifies end of source program with starting address of ALPHA.

6.9.3 GOTO

6.9.3.1 Format

Label field	Command field	Argument field
	GOTO,Exp	S1,S2....SN

6.9.3.2 Where

Label field -- not used
Command field -- The initial command field must contain the symbol "GOTO". CF(1) can contain an evaluatable expression or be void.
Argument field -- The argument field must contain one or more subfields, each of which contains a symbol.

6.9.3.3 Function

The GOTO directive causes DUAL to skip input statements until the designated symbol in the argument field of the GOTO statement is found in the label field of a subsequent input statement. The GOTO directive is a process time function only. That is, it is used to direct the sequence of input statement processing. When DUAL is searching for the statement whose label corresponds to the label of the GOTO, it operates in a skipping mode during which it ignores all statements except those which terminate previously existing processing modes (see Appendix A). A void CF(1) designates the initial symbol in the argument field is to be used. Otherwise, the value in CF(1) designates which symbol is to be used from the argument field (where 0=AF(0), 1=AF(1), etc.).

A blank skip-to label will cause the "GOTO" line to be ignored. If the directive is within a META, and the skip-to label is a name of a META, then the directive will act as a "META CALL LINE" with the current META's arguments being passed to the called META. Within a META the skip-to label may also be the name of a META on the selective meta library (if the selective META facility is available), unless the label begins with a $ indicating a locally defined label.

6.9.3.4 Example Usage

Assume ABC currently has the value 4.

Label field	Command field	Argument field
	GOTO,5-ABC	ALPHABET,ALPHA

Causes DUAL to enter the skipping mode until a subsequent line is found with a label "ALPHA".

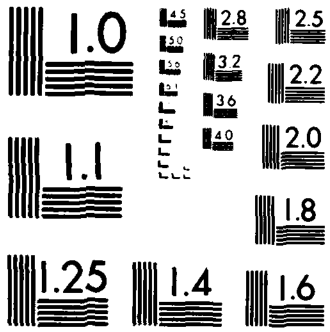
6.9.4 JUMPSYM

6.9.4.1 Format

Label field	Command field	Argument field
	JUMPSYM	C.S.,Relational, C.S.,Label

AD-A150 584 PROCEEDINGS OF THE TECHNICAL FORUM (3RD) ON THE F-16 6/6
MIL-STD-1750A MICROP..(U) AERONAUTICAL SYSTEMS DIV
WRIGHT-PATTERSON AFB OH J L PESLER ET AL. 06 MAY 82
UNCLASSIFIED ASD-TR-82-5011-VOL-2 F/G 9/2 NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

6.9.4.2 Where

Label field -- not used
Command field -- JUMPSYM
Argument field -- The argument field must contain four subfields whose contents are:

AF(0) - A character string that is either a symbol, a number or a field function.
AF(1) - Either "EQ" or "NQ" relational.
AF(2) - Subfield description same as AF(0).
AF(3) - A legal symbol (GOTO label).

6.9.4.3 Function

The JUMPSYM directive enables the user to conditionally determine the sequence in which input statements are processed. The initial argument subfield, AF(0), is compared against AF(2), and then based on the relational in AF(1), DUAL will continue its current processing mode or enter the skip mode (see Appendix A for DUAL Processing Modes). The JUMPSYM directive is a powerful tool for scanning arguments within a META call line and then generating the proper object (by directing the sequence of input line processing) based on the results of the scan.

A blank skip-to label will cause the line to be ignored. If the directive is within a META definition and the skip-to label is a name of a META, then the directive will act as a "META CALL LINE" with the current META's arguments being passed to the called META. Within a META the skip-to label may also be the name of a META on the selective META library (if the selective META facility is available), unless the label begins with $ indicating a locally defined label.

6.9.4.4 Error Conditions

- Argument field has less than four subfields
- Illegal symbol
- Illegal relational

6.9.4.5 Example Usage

Label field	Command field	Argument field
	JUMPSYM	ALPHAX,EQ,CF(1),XYZ

Assuming this line is within a META, then if CF(1) is equal to "ALPHAX", this statement would change the processing mode to skip-to until "XYZ" is found in the label field of a subsequent statement.

	JUMPSYM	AF(0),NQ,VOLTS,$1

Assuming this line is within a META, then if the initial argument field of the calling line for this META was not equal to "VOLTS", DUAL would go into the skipping mode until it found $1 in a subsequent input statement label field.

6.9.5 JUMPVAL

6.9.5.1 Format

Label field	Command field	Argument field
	JUMPVAL	Exp,Rel,Exp,Symbol

6.9.5.2 Where

Label field -- not used
Command field -- JUMPVAL
Argument field -- The argument field must contain four subfields whose contents are:

AF(0) - Evaluatable expression
AF(1) - Relational symbol (see Section 4)
AF(2) - Evaluatable expression
AF(3) - Symbol (GOTO label)

6.9.5.3 Function

The JUMPVAL directive allows the user to conditionally determine the sequence in which input statements are processed. AF(0) and AF(2) are evaluated as expressions and then are compared based on the relational in AF(1). If the relationship between AF(0) and AF(2) is as specified by the relational, then DUAL will enter the skip mode until the symbol in AF(3) is found in a subsequent input statement's label field. The JUMPVAL directive is a powerful tool for evaluating arguments within a META call line and then generating the proper object (by directing the sequence of input line processing) based on the results of the evaluation.

A blank skip-to label will cause the line to be ignored. If the directive is within a META definition and the skip-to label is a name of a META, then the directive will act as a "META CALL LINE" with the current META's arguments being passed to the called META. Within a META the skip-to label may also be the name of a META on the selective META library (if the selective META facility is available), unless the label begins with a $ indicating a locally defined label.

6.9.5.4 Error Conditions

- Illegal expression
- Illegal relational
- Illegal symbol

6.9.5.5 Example Usage

Label field	Command field	Argument field
	JUMPVAL	A+B,GT,100,$50

If the expression A+B is greater than 100, DUAL will enter the skip mode looking for label "$50".

	JUMPVAL	NUM(2),LT,5,$107

If the number of subfields in the argument field is less than 5, this statement will change the processing mode.

	JUMPVAL	AF(1),GQ,10,ABC

Assuming this line is within a META, then if argument subfield one is greater than or equal to 10, DUAL will enter the skip mode and will skip statements until symbol "ABC" is found in a subsequent input statement input statement's label field.

	JUMPVAL	AF(0),EQ,CF(3),123

This statement is illegal since there is no skip symbol in AF(3).

6.9.6 VOID

6.9.6.1 Format

Label field	Command field	Argument field
	VOID	Subfield,label,label

6.9.6.2 Where

Label field -- not used
Command field -- VOID
Argument field -- The initial argument subfield is used to check a particular subfield for being void. AF(1) designates where to go if the designated subfield is void and should contain a symbol or be blank. AF(2) designates where to go to if the designated subfield is not void and should contain a symbol or be blank. A blank goto label will cause DUAL to just continue processing in its current mode.

6.9.6.3 Function

The VOID directive is used to check a particular subfield for being void. The VOID directive can be used to skip a series of lines based on the presence or absence of a particular subfield from a META call.

A blank skip-to label will cause the line to be ignored. If the directive is within a META definition and the skip-to label is a name of a META, then the directive will act as a "META CALL LINE" with the current META's arguments being passed to the called META. Within a META the skip-to label may also be the name of a META on the selective META library (if the selective META facility is available), unless the label begins with a $ indicating a locally defined label.

6.9.6.4 Error Conditions

- Illegal GOTO symbol

6.9.6.5 Example Usage

Label field	Command field	Argument field
MODIFY	META	
LF(0)	LOAD	AF(0)
	VOID	AF(2),$1
	ADD	AF(1)
	STORE	AF(2)
	AMEND	
$1	LABEL	
	STORE	AF(1)
	MEND	

The MODIFY META expects either 2 or 3 arguments. Where there are only 2 arguments (AF(2) is void), then a LOAD-STORE set of instructions is generated. When there are 3 arguments (AF(2) is not void), then a LOAD-ADD-STORE set of instructions is generated.

```
MODIFY          ALPHA,GAMMA,BETA
```

Sets BETA=ALPHA+GAMMA.

6.9.7 IF

6.9.7.1 Format

Label field	Command field	Argument field
	IF	Expression

6.9.7.2 Where

Label field -- not used
Command field -- IF
Argument field -- an evaluatable expression

6.9.7.3 Function

The IF command is used to conditionally assemble sequences of code. The IF directive works in conjunction with the ENDIF and ELSE directive. If the expression in AF(0) is zero, then subsequent source lines are skipped until the corresponding ENDIF or ELSE directives are encountered. If the expression in AF(0) is non-zero, then subsequent source lines are assembled until an ELSE or ENDIF directive is encountered. IF's may not be nested (unless indirectly by calling another META with an IF), and cannot go beyond the boundaries of other process control directives (e.g., META/MEND, LOOP/LOOPTEST, etc.).

6.9.7.4 Example Usage

Send ENDIF directive.

6.9.7.5 Note

The IF/ELSE/ENDIF directives should not be used in conjunction with the GOTO type directives.

6.9.8 ELSE

6.9.8.1 Format

Label field	Command field	Argument field
	ELSE	

6.9.8.2 Where

Label field -- not used
Command field -- ELSE
Argument field -- not used

6.9.8.3 Function

The ELSE directive is used in conjunction with the IF and ENDIF directives to conditionally assemble sequences of code. The ELSE directive reverses the logic of its respective IF command.

6.9.8.4 Example Usage

See ENDIF directive.

6.9.9 ENDIF

6.9.9.1 Format

Label field	Command field	Argument field
	ENDIF	

6.9.9.2 Where

Label field -- not used
Command field -- ENDIF
Argument field -- not used

6.9.9.3 Function

The ENDIF directive works in conjunction with the IF and ELSE directives to conditionally assemble sequences of code. The ENDIF directive must have a corresponding IF. The ENDIF directive terminates an IF condition.

6.9.9.4 Example Usage

```
(1)                IF             ALPHA.GT.10
          XYZ      DATA           5
          ABC      DATA           10
                   ELSE
          XYZ      DATA           6
          ABC      DATA           7
                   DATA           0
                   ENDIF
```

If ALPHA is greater than ten, then XYZ and ABC will reference data cells containing 5 and 10 respectively. Otherwise, XYZ and ABC will reference data cells containing 6 and 7, and another data cell with a value of zero will be generated.

```
(2)        IF      1     This will always be true.
           DATA    5     This will always be assembled.
     ABC   EQU     5     This will always be assembled.
           ELSE          Reverses IF logic.
           DATA    5     This will be skipped.
           ENDIF         Ends IF logic.
```

6.9.10 IFS

6.9.10.1 Format

Label field	Command field	Argument field	General field 3	General field 4
	IFS	char.string	char.string	goto - label

6.9.10.2 Where

Label field -- not used
Command field -- IFS
Argument field -- character string
General field 3 -- character string
General field 4 -- goto label

6.9.10.3 Function

The IFS directive is used to compare two text strings. If the strings are identical, then subsequent lines are skipped until the goto label is encountered. The IFS directive differs from the JUMPSYM in that the JUMPSYM directive requires that the arguments be legal symbols.

6.9.10.4 Example

```
QUESTION        META    3
QMARK           SET     0
                IFS     GF(2) ? QMARK
                AMEND
QMARK           SET     1
                MEND
```

6.10 META DEFINITION DIRECTIVES

6.10.1 GENERAL

A META is a set of input statements starting with a META directive and ending with a MEND directive. A user invokes (calls) a particular META by referencing in the initial command subfield the label supplied on a META directive. The remaining fields and subfields are used to supply arguments for the META.

The arguments from a META call line are referenced within a META via the field functions and the NUM function.

A particular META may call other META's or may call itself (recursive). No limitation is placed on the user as far as depth of recursion or level of nested META calls. A META must be defined prior to its being called. META's may be defined anywhere within a source program.

Since a META call invokes the processing of a series of input statements, the user can specify which lines of the META are to be output on the listing output device. If the META's are used to define a procedure-oriented language, then this feature allows the user the capability of displaying or suppressing the META expansion (possibly machine instructions) that is generated with each META call by simply varying one input statement and parameterizing the META print indicator.

6.10.2 META

6.10.2.1 Format

Label field	Command field	Argument field
Symbol	META	Expression

6.10.2.2 Where

Label field -- The label field must contain a symbol. This defines the name of the META.
Command field -- META
Argument field -- The initial argument subfield must contain an evaluatable expression which results in a value between 0 and 6 inclusive. Each value is used to control the listing and output of input statements within the META as follows:

0 = No listing.
1 = List all input statements.
2 = List only those statements which generate object or equate a symbol to a value.
3 = List only those statements which affect the location counter.
4 = Overlap printout for META's which generate a single word or object.
5 = Printout from within a META under LISTING control directive specification. The printout occurs at META processing termination (MEND,AMEND, or MENDED).
6 = Overlap printout from within a META under LISTING control directive specification. The printout occurs at META processing termination (MEND, AMEND, or MENDED).

6.10.2.3 Function

The META directive is used to define a META. The reader is advised to see Section 7 for a more detailed discussion of META usage.

6.10.2.4 Error Conditions

- Illegal symbol

6.10.2.5 Example Usage

Label field	Command field	Argument field
MOVE	META	1
LF(0)	CLA	GF(2)
	STA	GF(3)
	MEND	

Defines a META named "MOVE". This META when called will generate a "CLA" instruction of the first argument subfield and a "STA" instruction of the second argument subfield (assuming "CLA" and "STA" were previously defined as instructions). If a label appears on the "MOVE" input statement, then it will be defined as equal to the current location counter.

```
                MOVE        ALPHA BETA
```

This statement would generate "CLA ALPHA" and "STA BETA" instructions.

```
    LOOP$TO     MOVE        SPEED VELOCITY
```

This statement would generate a "CLA SPEED" and a "STA VELOCITY" instruction. The label "LOOP$TO" would be defined as equal to the location of the "CLA SPEED" instruction.

See Section 7 for additional examples.

6.10.3 MEND

6.10.3.1 Format

Label field	Command field	Argument field
	MEND	Symbol(opt.)

6.10.3.2 Where

Label field -- not used
Command field -- MEND
Argument field -- The argument field may contain a symbol which will be used as a "GOTO LABEL" after the end of the META call.

6.10.3.3 Function

The MEND directive signifies the end of a META definition or call. If a symbol is present in the argument field, then a return from a META call will cause the "GOTO MODE" to be entered after the return. The MEND directive will be printed based on the "PRINT STATUS" prior to the call when terminating a META call.

6.10.3.4 Error Conditions

- A processing mode other than META call or META save (i.e., a GOTO within a META that is not satisfied prior to or at the MEND).

6.10.3.5 Example Usage

Label field	Command field	Argument field
ABC	META	1

Defines new META name "ABC".

.
.
.

	MEND	

This statement signifies the end of META "ABC".

See Section 7 for additional examples.

6.10.4 AMEND

6.10.4.1 Format

Label field	Command field	Argument field
	AMEND	Symbol(optional)

6.10.4.2 Where

Label field -- not used
Command field -- AMEND
Argument field -- The argument field may contain a symbol which will be used as a "GOTO LABEL".

6.10.4.3 Function

The AMEND directive signifies the end of a META call. If a symbol is present in the argument field, then a return from a META call will cause the "GOTO MODE" to be entered after the return. The AMEND directive will be printed based on the "PRINT STATUS" prior to the call when terminating a META call.

6.10.4.4 Example Usage

Label field	Command field	Argument field
MOVE	META	1
	VOID	CF(1),$1
LF(0)	INDEX	CF(1)-1
	CLA	AF(0),X
	STA	AF(1),X
	TIX	$-2
	AMEND	
$1	LABEL	
LF(0)	CLA	AF(0)
	STA	AF(1)
	MEND	
	MOVE	A,B

This line would generate instructions to move a value from A to B.

	MOVE,10	ALPHA,BETA

This line would generate instructions to move 10 words from ALPHA to BETA.

6.10.4.5 Special Comments

A META may have any number of AMEND statements, but can have only one MEND statement.

6.10.5 AMETA

6.10.5.1 Format

Label field	Command field	Argument field
Symbol	AMETA	Expression

6.10.5.2 Where

Label field -- The label field must contain a symbol. This defines the name of a META.
Command field -- AMETA
Argument field -- The initial argument must contain a evaluatable expression to be used in controlling the listing and output of lines from within a META (see description of META argument field for more details).

6.10.5.3 Function

The AMETA directive is used in conjunction with the MENDED directive for nested meta definitions. Thus, a META may be defined within a META. Via the AMETA-MENDED directives, a series of METAS can be developed which generate other METAS.

6.10.5.4 Error Conditions

- Illegal symbol

6.10.5.5 Example Usage

Label field	Command field	Argument field	Comments
METADEF	META		START OF META METADEF
GF(2)	AMETA	3	DEFINE NEW META
	.....		BODY OF META
	MENDED		END OF NEW META
	MEND		END OF METADEF
PERSONNEL	METADEF	PAYROLL	INCOME TAX DEDUCTIONS

This would cause a new META named PAYROLL to be defined.

6.10.6 MENDED

6.10.6.1 Format

Label field	Command field	Argument field
	MENDED	Symbol (option)

6.10.6.2 Where

Label field -- not used
Command field -- MENDED
Argument field -- The argument field may contain a symbol which will be used as a "GOTO LABEL" after the end of the META call.

6.10.6.3 Function

The MENDED directive is used in conjunction with the AMETA directive to signify the end of a nested meta definition. For further explanation, see both the AMETA and META directives.

6.10.7 DUALMETA

6.10.7.1 Form

Label field	Command field	Argument field
	DUALMETA	Exp1,Exp2,Exp3

6.10.7.2 Where

Label field -- not used
Command field -- DUALMETA
Argument field -- The argument field consists or three subfields, each of which is optional, and if specified must contain an evaluatable expression which results in a value of 0 or 1.

AF(0) = Processing indicator for selective metas whose status is temporary

0 = Temporary, 1 = Permanent

AF(1) = Processing indicator for selective metas whose status is permanent.

0 = Temporary, 1 = Permanent

AF(2) = The status to be assigned metas during their pre-processing for the selective meta library.

0 = Temporary, 1 = Permanent

6.10.7.3 Function

The DUALMETA directive enables the user to dynamically control the processing of selective metas, and to designate the status of metas during their pre-processed generation as members of the selective meta library. Selective metas processed as temporary are not resident in core for the duration of the assembly. Therefore, they are retrieved from the selective meta library each time they are called for processing. Selective metas processed as permanent are resident in core for the duration of the translation; therefore, retrieval from the selective meta library is performed only once when and if the meta is called.

6.10.7.4 Default Case

DUAL will process selective metas according to the status indication resident in the selective meta library.

DUAL will assign a status of temporary during the pre-processing of metas for selective meta library generation.

6.10.7.5 Error Conditions

- Illegal expression in argument subfield
- Argument subfield values not equal to 0 or 1 or void

6.10.7.6 Example Usage

1. Retrieval from the selective meta library

Assume the selective meta library contains the following metas as members:

LOAD (Temporary status)
MOVE (Permanent status)
STORE (Permanent status)

Label field	Command field	Argument field
	LOAD	ALPHA

The selective meta LOAD will be brought into core, processed, then deleted from memory. This is default processing of selective meta library members with temporary status.

	MOVE	ALPHA,BETA

The selective meta MOVE will be brought into core, defined in the command and symbol table and then processed. All subsequent calls for meta MOVE will cause no access from the selective meta library since the MOVE meta is resident in core for the duration of the translation. Therefore, MOVE is not affected by subsequent DUALMETA statements. This is default processing of selective meta library members with permanent status.

```
                    DUALMETA        1,0
```

This reverses the temporary/permanent processing of selective metas.

```
                    LOAD            ALPHA
```

The selective meta LOAD will now be processed as if its status were permanent. That is, it will be defined in the command and symbol table and remain resident in core.

```
                    STORE           BETA
```

The selective meta STORE will now be processed as if its status were temporary. That is, it will not be defined in the command and symbol table and will not remain in core following its processing.

2. Pre-processing metas for selective meta library generation.

During this mode of operation metas defined in the source input stream will be processed only for selective meta library generation. That is, they will not be defined in the command and symbol table, and, therefore, are not available for meta processing.

```
        Label           Command         Argument
        field            field           field

        LOAD            META            3
                        L               5,AF(1)
                        MEND
```

The LOAD meta will be output to the selective meta library in DUAL internal format. The status assigned to this meta is temporary which is the default status.

```
                    DUALMETA        ,,1
```

This designates pre-processed selective metas to be assigned the status of permanent on the selective meta library.

```
MOVE        META        MPI
            L           5,AF(1)
            ST          5,AF(2)
            MEND
```

The meta MOVE will be output to the selective meta library with a status of permanent.

6.10.7.7 Special Comments

This directive is applicable only when the selective meta feature is active for the installation. Certain installations have restrictions as to the length of the meta name (e.g., the PDP-11 has a six character maximum). It is the user's responsibility to adhere to installation restrictions, since the DUAL processor is not cognizant of installation dependencies.

6.10.8 METAPRINT

6.10.8.1 Format

Label field	Command field	Argument field
	METAPRINT	Expression

6.10.8.2 Where

Label field -- not used
Command field -- METAPRINT
Argument field -- An evaluatable expression which results in a parameter value between 0 and 6.

6.10.8.3 Function

The METAPRINT directive enables the user to dynamically control the current META print indicator while a META is being processed. A METAPRINT directive can only be used with a META. As many METAPRINT directives can be used within a META as the user deems necessary. Note that the METAPRINT directive itself is never printed unless there is an error.

6.10.8.4 Error Conditions

- Illegal META print indicator
- Not within a meta

6.10.8.5 Example Usage

```
ABC        META        1        Print all lines
           METAPRINT 0          Turns off print
           METAPRINT 3          Now subsequent data generation
                                Lines will be printed
           METAPRINT 1          All subsequent lines will be
                                printed
           MEND
```

6.11 ERROR NOTIFICATION DIRECTIVES

6.11.1 GENERAL

One of the primary functions of METAs in DUAL is the generation of higher order languages. METAs defined by the programmer not only can generate different object code based on the arguments, but can also flag a statement as being in error. The ERROR directive causes the erroneous statement to be listed with the error flags and the error and line number to appear in the error summary at the end of the processing of the input statements.

6.11.2 ERROR

6.11.2.1 Format

Label field	Command field	Argument field
	ERROR	Expression

6.11.2.2 Where

Label field -- not used
Command field -- The initial command subfield must contain the symbol "ERROR".
Argument field -- The initial argument subfield must contain an evaluatable expression which results in a value between 500 and 9999 inclusively.

6.11.2.3 Function

The ERROR directive causes the ERROR statement to be flagged in error and the line and error number (from the expression) to be saved and output in the error summary. The ERROR directive provides a means for a META to inform a calling line of an error. Note that error numbers 1-499 are saved for DUAL and associated processors, and that error number 9999 does not increment the error counter nor will it be printed in the error summary.

6.11.2.4 Error Conditions

- Illegal expression

6.11.2.5 Example Usage

Label field	Command field	Argument field
	ERROR	813

Flag this line as an error. The line number for this line along with the number 813 will appear in the error summary.

6.11.3 WARNING

6.11.3.1 Format

Label field	Command field	Argument field
	WARNING	Expression

6.11.3.2 Where

Label field -- not used
Command field -- WARNING
Argument field --The expression may take any of the values 0,1, or 2. A value of 0 turns the WARNING operation off. A value of 1 results in a warning message being printed any time the value of a label is changed (default). A value of 2 results in a warning being printed whenever the value of a global label (only) is changed.

6.11.3.3 Function

The WARNING directive allows the user to monitor duplicate EQU redefinitions.

6.11.3.4 Example Usage

Label field	Command field	Argument field
	WARNING	0

This would cause all subsequent redefinitions to be ignored.

6.12 LOOP CONTROL DIRECTIVES

6.12.1 GENERAL

The LOOP CONTROL directives provide a tool for conditional and/or repetitive input statement processing. The LOOP CONTROL directives can be used for the generation of a series of values, or if within a META, as an aid in scanning through a series of subfields within a particular field.

6.12.2 LOOP

6.12.2.1 Format

Label field	Command field	Argument field
Symbol	LOOP	Exp,Exp,Exp

6.12.2.2 Where

Label field -- A legal symbol (if the symbol already exists, then it must have previously been a parameter or a loop index). Note that a local symbol found in error may cause subsequent errors.
Command field -- LOOP
Argument field -- The argument field must contain three subfields, each of which contains an evaluatable expression that results in a positive value. The contents of the argument subfields are:

AF(0) = Initial value of symbol.
AF(1) = Increment value for symbol. This must be greater than zero.
AF(2) = Test value for symbol. This must be at least as great as AF(0).

6.12.2.3 Function

The LOOP directive defines the beginning of an iteration function. The subsequent input statements up to and including the LOOPTEST statement will be processed as many times as defined by the initial conditions within the LOOP statement. The reader is advised to see the LOOPTEST directive for a more detailed explanation of the LOOP directive.

6.12.2.4 Error Conditions

- Illegal symbol
- Illegal expression

6.12.2.5 Example Usage

Label field	Command field	Argument field
I	LOOP	0,1,9

Defines the start of a loop which will initialize the symbol I to 0, increment it by 1, and terminate the loop when I exceeds 9 (ten times through loop).

6.12.3 LOOPTEST

6.12.3.1 Format

Label field	Command field	Argument field
	LOOPTEST	Symbol (opt.)

6.12.3.2 Where

Label field -- not used
Command field -- LOOPTEST
Argument field -- Optional symbol representing label of a subsequent statement to which control will be returned when the loop processing has been completed. If omitted, control will proceed to the next sequential statement.

6.12.3.3 Function

The LOOPTEST directive is used to signify the end of an iteration loop. To better understand the function of the LOOPTEST directives, a step-by-step account of their operation is described:

STEP 1 The symbol in the LOOP label field is defined as a parameter with an initial value of AF(0).

STEP 2 Information is saved containing the LOOP symbol, the increment and the terminating value.

STEP 3 Subsequent (after LOOP statement) input statements are processed until the LOOPTEST statement is found. When the LOOPTEST statement is found, then Step 4 is entered.

STEP 4 The LOOP symbol's value is incremented by the increment value saved in Step 2.

STEP 5 If the LOOP symbol's value is not greater than the termination value saved in Step 2, return to Step 3 and continue on from there.

STEP 6 End of loop.

6.12.3.4 Example Usage

Label field	Command field	Argument field
I	LOOP	0,1,9
	DATA	I
	LOOPTEST	

The three lines would generate 10 object words with values from 0 to 9.

ALPHA	EQU	$
J	LOOP	0,1,100
	DATA	0
	LOOPTEST	

These four lines would generate 101 object words all with a value of zero. The initial line could be referenced by the symbol "ALPHA".

```
SQUARES     META
            VOID        AF(0),$1
I           LOOP        0,1,NUM(2)-1
            DATA        AF(I)*AF(I)
            LOOPTEST
$1          MEND
```

These five lines define a META named squares. The META squares will generate object words with the squares of the arguments. The LOOP statement is used to index through the calling argument field. Note that a loop from 0,1,NUM(2)-1 will iterate through each subfield within the calling argument field.

```
            SQUARES     5,6,7
```

This statement would generate object values of 25, 36 and 49.

6.12.4 LOOPEXIT

6.12.4.1 Format

Label field	Command field	Argument field
	LOOPEXIT	Symbol (opt.)

6.12.4.2 Where

Label field -- not used
Command field -- LOOPEXIT
Argument field -- Optional symbol representing label of a subsequent statement (beyond the LOOPTEST) that will receive control. If omitted, control will proceed to the next sequential statement after the LOOPTEST.

6.12.4.3 Function

The LOOPEXIT directive provides a method for terminating an iteration loop prior to the final iteration.

6.12.4.4 Example Usage

Label field	Command field	Argument field
I	LOOP	0,1,NUM(2)-1

Start of iteration loop through arguments from a META call.

	JUMPVAL	AF(I),NQ,0,PRODUCT

Exit if an argument is zero.

	LOOPEXIT	

Exit loop.

PRODUCT	EQU	PRODUCT*AF(I)

Accumulate products of the arguments.

	LOOPTEST	

End of iteration loop.

6.13 LOOP GENERATION DIRECTIVES

6.13.1 GENERAL

A problem that currently exists in both meta assemblers and macro assemblers is that they are unable to generate higher order languages which contain a command for generating efficient object for looping at execution time, (i.e., like a FORTRAN "DO" statement or JOVIAL "FOR" statement). The problem resolves to the fact that at the time of processing the iteration command, the META writer knows exactly what to generate at the end of the loop, yet has no mechanism (directive) for saving these lines and then processing them at the proper time. DUAL's loop generation control directives provide a mechanism for the efficient generation of execution time loop commands. The loop generation directives allow the user to save a series of input statements and then have them processed at a later time.

6.13.2 WITHHOLD

6.13.2.1 Format

Label field	Command field	Argument field
	WITHHOLD	Symbol

6.13.2.2 Where

Label field -- not used
Command field -- WITHHOLD
Argument field -- The initial argument subfield must contain a symbol.

6.13.2.3 Function

The WITHHOLD directive changes DUAL's processing mode to WITHHOLD save. Subsequent input statements will be saved until a WEND statement is processed. These lines will be processed when the symbol in the WITHHOLD statement is found in the label field of a LABEL statement that is not within a META.

6.13.2.4 Error Conditions

- Illegal Symbol

6.13.2.5 Example Usage

Label field	Command field	Argument field
	WITHHOLD	ALPHA

This statement would cause subsequent lines to be withheld with symbol "ALPHA" being used to determine when the lines are to be processed.

For additional examples, see WEND directive.

6.13.3 WEND

6.13.3.1 Format

Label field	Command field	Argument field
	WEND	

6.13.3.2 Where

Label field -- not used
Command field -- WEND
Argument field -- not used

6.13.3.3 Function

The WEND directive is used to signify the end of a withhold save process mode. (See Appendix A - DUAL Processing Modes).

6.13.3.4 Example Usage

Assume a language is being developed with an iteration command called REPEAT. This command is to generate object to control a loop. The format for REPEAT is to be:

Label field	Command field	Argument field
Symbol	REPEAT	Symbol,Symbol,Exp,Exp,Exp

Where:

LF(0) = Repeat start symbol
CF(0) = REPEAT
AF(0) = Repeat termination symbol
AF(1) = Repeat index symbol
AF(2) = Initial value for repeat index
AF(3) = Increment value for repeat index
AF(4) = Termination value for repeat loop

Then the following META might be used to implement the REPEAT command. This example assumes the following target instructions:

LDX = Load index with contents of designated address.

CLA = Load accumulator with contents of designated address.

STA = Store accumulator into contents of designated address.

COM = Skip an instruction if accumulator is greater than contents of designated address.

INX = Increment index by contents of designated value.

TRA = Jump to designated location.

Label field	Command field	Argument field	Comment field
REPEAT	META	1	Defines a META named Repeat.
	LDX	L(AF(2))	Set index to initial value.
	CLA	L(AF(2))	These two statements define the initialization of the index.
	STA	AF(1)	
LF(0)	EQU	$	This statement would define the starting location for the statements that are to be repeated.
	WITHHOLD	AF(0)	Would cause the following lines to be withheld until the WEND is processed.
	CLA	AF(1)	Gets current value of index into accumulator
	ADD	L(AF(3))	Increments index symbol.
	STA	AF(1)	Resets index value symbol.
	INX	L(AF(3))	Increment Index.
	COM	L(AF(4))	Checks index.
	JUMP	LF(0)	Perform next iteration.
	WEND		End of withhold mode.
	MEND		End of REPEAT meta.

Label field	Command field	Argument field
LOOPBEG	REPEAT	LOOPEND,LOOPIX,0,1,9

Now the preceding REPEAT statement would generate the following instructions:

```
                LDX        L(0)
                CLA        L(0)
                STA        LOOPIX
LOOPBEG         EQU        $
```

which should perform the loop initialization. At the termination point of the loop the user would write:

```
LOOPEND         LABEL
```

This statement should cause the lines that are being withheld to generate the following:

```
                CLA        LOOPIX
                ADD        L(1)
                STA        LOOPIX
                INX        L(1)
                COM        L(9)
                JUMP       LOOPBEG
```

which would control the updating of indexes and the termination of the loop.

6.13.3.5 Special Comments

The above example illustrates the language generation power of METAs. The loop generation command, which is usually one of the most complex parts of a language processor, was implemented in a fraction of the time spent with conventional methods. It should be noted that the above META does not have any error tests. For a more complete REPEAT META, questions should have been asked concerning the number of argument fields, whether the label field contains a symbol, if the termination value for the loop is greater than the initial value, etc. These questions could have been asked and appropriate error messages generated by using the JUMPSYM, JUMPVAL and ERROR directives.

CHAPTER 7

Meta Usage

7.1 GENERAL

This section provides the reader with an idea of some of the various usages of METAs. The user of DUAL is provided with the most powerful set of directives and functions yet devised for language translation. METAs enable the programmer to generate different sequences of code according to conditions at the time of processing.

METAs are user-defined commands to the META-processor. They perform like directives -- only at a higher level. Directives are "built-in" and are given predefined meanings that are universally applicable to all language processing. METAs instruct the META-processor to perform special process-time operations in response to particular attributes of the source input information.

METAs are constructed as open process-time subroutines. They may call other METAs to any level, including recursive calls on themselves. In this manner it is easy for the user to modularize his descriptions of an assembly, compilation or translation process.

Directives are the fundamental units of instructions in a META. They are combined by the META-programmer in a manner that describes the mapping function between source and object.

As a newly defined source language receives application usage, certain inherent strengths and weaknesses will become apparent. It is possible for the META-programmer to add or delete commands from a META-defined source language with a minimum of redefinition: He simply adds or removes those METAs that are applicable. Because of this feature, languages created via a META-processor are open-ended.

7.2 Initialization

A META can be developed to perform the initialization of various system parameters to be used during the input statement processing. The META, for example, might accept as input a number which indicates the target computer, and then perform the proper initialization. Below is an example of just such a META.

Label field	Command field	Argument field
INIT	META	0
	JUMPVAL	AF(0),NQ,7094,$1
	SIZE	36
	KSIZE	32767
	AMEND	
$1	JUMPSYM	AF(0),NQ,SIGMA,$2
	SIZE	32
	KSIZE	131071
	AMEND	
$2	ERROR	752,(ILLEGAL TARGET)
	MEND	

7.3 Data Generation

Computer programs often contain preset data within a program. If the data is part of a table, whose information format is somewhat the same (except for contents), then a META can be developed which will greatly decrease the burden of coding along with the probability of making an error. For example, assume a program needs the following data for 100 employees of a given corporation:

1. Initials of name
2. Man number
3. Age in years
4. Marital status (0 = not married, 1 = divorced, 2 = widowed, 3 = married)
5. Number of dependents
6. Hourly salary

and further assume the data is to be placed in an 18-bit computer as follows:

```
0.........................5..6.......................17
.....................Initials of name...................
.........................Man number.....................
Marital status                     Age
Number dependents                  Hourly salary
```

For employee JFK, the following four statements might be used (assuming L1 is a LIST 6,12)

```
1.    DATA      HOL(JFK)
2.    DATA      10732
3.    GEN,L1    3,45
4.    GEN,L1    8,690
```

These four statements would then have to be repeated for each employee. This is the conventional way of approaching the problem in current language processors. In DUAL, this problem could be solved by defining the following META:

Label field	Command field	Argument field	Comments
EMPLOYEE	META	3	
	DATA	HOL(GF(2))	NAME
	DATA	GF(3)	Man No.
	GEN,L1	GF(4),GF(5)	Married, Age
	GEN,L1	GF(6),GF(7)	Dependents, Salary
	MEND		

and would be called as follows:

```
EMPLOYEE      JFK      10732 3 45 8 690
```

This META would, in effect, cause the writing of a single input statement for each additional employee (as opposed to four in the conventional method). If should be noted that the META could also contain checks for legal man numbers, realistic ages, etc., if the META writer had deemed it necessary.

7.4 Instruction Expansion

Another usage of METAs is for instruction expansion. That is, for either changing the name of an instruction or having the META generate a series of instructions which are used repetitively throughout the program.

An example of this might be a MOVE META. Instead of continually writing a load and store instruction when it is required to move data from one location to another, one could develop the following META:

```
        Label          Command        Argument
        field           field          field

        MOVE           META           3
        LF(0)          LOAD           AF(0)
                       STORE          AF(1)
                       MEND
```

then instead of writing

```
                       LOAD           ALPHA
                       STORE          BETA
```

the programmer could code it as

```
                       MOVE           ALPHA,BETA
```

Another example might be a subroutine call with arguments.

Normally one might call a subroutine as

```
                       BAL            SCHEDULER
                       DATA           2
                       DATA           ALPHA
                       DATA           BETA
```

One approach would be to have the following META:

```
        CALL           META           3
        LF(0)          BAL            AF(0)
                       VOID           AF(1),$1
                       LOOP           1,1,NUM(2)-1
                       DATA           AF(I)
                       LOOPTEST
        $1             MEND
```

Then the programmer would code the subroutine call as:

```
                       CALL           SCHEDULER,2,ALPHA,BETA
```

and would generate the same object as above.

7.5 Instruction Generation

A new approach to a multi-computer environment which DUAL makes possible is that of a super instruction set, which, when utilized with the proper METAs, is translated into the proper object. This approach would allow machine-like language programs to be written for more than just one computer. Two possible ways of accomplishing this task are:

- Calling the proper system from a library tape for the particular computer to be used. This would mean that the library tape would contain a set of METAs and instruction definitions for each particular computer. The programmer would then just alter his LIBRARY in statement at the front of this program to be able to translate his program to the proper target machine.

- Developing instruction METAs that ask what is the target machine and then generating the object accordingly.

This first step in either of these methods is the definition of a computer that will provide the following:

- Ease of instruction translation to any specific target machine.

- Able to produce efficient programs that are easy to implement.

For example, assume our theoretical computer is to have a memory increment instruction named "BUMP". The format for BUMP is to be as follows:

Label field	Command field	Argument field
Symbol (opt.)	BUMP	Symbol.Expression

For some computers this might involve a load, add and store instruction. For other computers this might be accomplished by a load and then a memory add instruction, and yet for other computers this might be accomplished in a single increment instruction. Assume an environment with two computers, X and Y, and that to accomplish the BUMP instruction the following is required:

COMPUTER X		COMPUTER Y	
1)	CLA	1)	INK (if value is less than 16)
	ADD		or
	STA	2)	CLA
			MADD

then the following META could be used assuming symbol "MACH" is equal to zero when translating for machine X.

```
Label         Command        Argument
field          field          field

BUMP          META           3
              JUMPVAL        MACH,NQ,O,$COMPY
LF(O)         CLA            AF(O)
              ADD            L(AF(1))
              STA            AF(O)
              AMEND
$COMPY        JUMPVAL        AF(1),GT,15,$2
LF(O)         INK            AF(O),AF(1)
              AMEND
$2            LABEL
LF(O)         CLA            L(AF(1))
              MADD           AF(O)
              MEND
```

7.6 Language Generation

One the major purposes of METAs is as a tool in the generation of user oriented languages. A user can develop a language to meet his specific data processing requirements, whether it be for matrix manipulation, automated checkout, banking, or whatever. The development of a META language is discussed in Appendix F.

7.7 Language Translation/Conversion

Often it is necessary to convert a program that was written on one computer to operate on another. This can be done by a series of METAs whose names are those of the original computer, and yet whose contents contain the proper instructions for the target machine.

A simple case would involve just the changing of the instruction name and the argument order. For example, assume a machine X and a machine Y each have an instruction that loads a general register

```
        Machine X________________

        CLA                          Address,Index,General register

        Machine Y________________

        LOAD,General register        Address,Index
```

then, to translate the machine X instruction to the machine Y instruction, the following META could be written with a LOAD instruction definition (see STRUCTURE - INST directive).

```
        CLA         META            3
                    PUNCH
                    LOAD,AF(2)      AF(0,AF(1)
                    PEND
                    MEND
```

The aforementioned usages of METAs are intended to illustrate some of the possible ways the programmer can use METAs. METAs may be used for any application the programmer desires, with the only limitation being that of the programmer's imagination.

CHAPTER 8

Listing Output

8.1 General

DUAL provides the user four distinct listings. The first is a listing of his source program, the second is a listing of the errors that occurred during the translation of the user's program, the third is a dictionary of global symbols and set/use (cross references), and the fourth is the range reference summary. The format for these listings is described below.

8.2 Source Line Listing Output

The listing of the source program occurs in parallel with the processing of the source input statements. The information that is provided for the source line listing output is explained below in the default order in which it appears. The user can modify the order and what is to be listed via the LISTING directive.

1. Error indicator -- any statement that is found in error will include the error indicator(*). The line and error number will also appear in the error notification listing.

2. Line number -- this is the number for source lines that are actually listed. If a line is in error, DUAL will save this number for the error notification listing.

3. Location offset -- for those input statements which effect the location counter (i.e., - DATA,GEN,machine instructions, ORIGIN, etc.), the contents of the location counter offset will be displayed.

4. Location section number -- current section location number.

5. Skip flag -- those lines that are skipped due to a GOTO type directive, will have a skip flag (*SKIP*).

6. Object or symbol value -- for those statements that generate object code (e.g., - DATA,GEN,machine instructions), or that equate symbols to a value, (EQU) the resulting value will be displayed.

7. Object (symbol) value classification flag -- a flag is provided describing the type of value that is displayed. If the object being displayed is part of an instruction or GEN directive, then one flag refers to the rightmost bits. The different classification flags are described below:

   0-31 = address section number
   F = forward reference
   X = external reference
   P = parameter
   A = absolute

8. Source line -- the source line is displayed here. If the line is from a META during a META call then the following statements hold with respect to the displaying of the source line (see TABSET directive):

   A) The label field is displayed in column 1-13 of the source line.

   B) The command field is displayed in columns 15-28 of the source line.

   C) The argument field is displayed in columns 30-80 of the source line.

   D) A maximum of 72 characters is displayed for any statement.

In addition to the above information, DUAL provides control for titles, subtitles, headings, spacing and page ejection during the listing of the source program.

8.3 Error Notification Listing Output

If there were errors during the processing of source input statements, DUAL will then inform the user of the line number on which the error occured and the error number (see Appendix B, Error Number Descriptions) associated with that line. There is no limit to the number of errors that may occur on a single line. The listing is output in the order in which the errors occured. A sentence stating the total number of errors that occured during the processing of the source program is output prior to the error notification listing.

8.4 Dictionary Listing Output

At the end of each source program DUAL provides the user with a symbol dictionary. Each global symbol is listed in the dictionary with the following information:

- Name of symbol
- Value
- Classification of type

0-31	=	Address section number
P	=	Parameter
X	=	External reference
U	=	Undefined
L	=	List

- References

An optional cross reference listing is provided, if requested. The cross reference is either by line number (default) or by location (via BYLOC on .DUAL statement).

8.5 Range Reference Listing

At the end of each source program DUAL provides the user with a listing of all expressions falling within a user specified range specification. For each range statement, a separate printout is made for all expressions which contain an address within the bounds of the range.

END

FILMED

3-85

DTIC